# Memory-Constrained Computing

**A THESIS**
**SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL**
**OF THE UNIVERSITY OF MINNESOTA**
**BY**

**Jeremy Iverson**

**IN PARTIAL FULFILLMENT OF THE REQUIREMENTS**
**FOR THE DEGREE OF**
**DOCTOR OF PHILOSOPHY**

**Dr. George Karypis, Advisor**

**November, 2017**

# Acknowledgements

First and foremost, I would like to thank God for everything.

There is no one who has contributed more to my graduate study and this thesis than my adviser, George Karypis. His seemingly endless patience with me is only equaled by his knowledge of and passion for Computer Science. Both of these characteristics have served as source of inspiration throughout my time as his advisee. I am very grateful for all of the opportunities that he has afforded me and all that I have learned from him.

I would like to thank Professors Jon Weissman, Sean Garrick, and Antonia Zhai for taking the time to serve on my thesis committees. I appreciate their valuable feedback, advice and encouragement.

I am thankful to all of my family, especially those who I repeatedly told that I was "about" to graduate for many years. Not once did they have anything to say but words of encouragement. This would not have been possible without their unconditional love and support.

I am especially indebted to the members of Karypis Lab: Asmaa, Agoritsa, Chris, David, Dominique, Eva, Fan, Haoji, Keven, Mohit, Rezwan, Santosh, Sara, Saurav, Shaden, Xia, Yevgeniy, and Zhonghua. Your passion for the pursuit of the elusive Ph.D. was inspiring and your kindness to lend an ear on so many occasions was invaluable. I am truly grateful to have worked with such dedicated and capable colleagues.

I would like to thank the staff at the Department of Computer Science, the Digital Technology Center, and the Minnesota Supercomputing Institute at the University of Minnesota for providing assistance, facilities, and other resources for my research.

Lastly, I would like to thank my wife, Leandra, and my children. There are no words to express my gratitude for everything that you have given to me.

## Abstract

The growing disparity between data set sizes and the amount of fast internal memory available in modern computer systems is an important challenge facing a variety of application domains. This problem is partly due to the incredible rate at which data is being collected, and partly due to the movement of many systems towards increasing processor counts without proportionate increases in fast internal memory. Without access to sufficiently large machines, many application users must balance a trade-off between utilizing the processing capabilities of their system and performing computations in memory. In this thesis we explore several approaches to solving this problem.

We develop effective and efficient algorithms for compressing scientific simulation data computed on structured and unstructured grids. A paradigm for lossy compression of this data is proposed in which the data computed on the grid is modeled as a graph, which gets decomposed into sets of vertices which satisfy a user defined error constraint $\epsilon$. Each set of vertices is replaced by a constant value with reconstruction error bounded by $\epsilon$. A comprehensive set of experiments is conducted by comparing these algorithms and other state-of-the-art scientific data compression methods. Over our benchmark suite, our methods obtained compression of 1% of the original size with average PSNR of 43.00 and 3% of the original size with average PSNR of 63.30. In addition, our schemes outperform other state-of-the-art lossy compression approaches and require on the average 25% of the space required by them for similar or better PSNR levels.

We present algorithms and experimental analysis for five data structures for representing dynamic sparse graphs. The goal of the presented data structures is two fold. First, the data structures must be compact, as the size of the graphs being operated on continues to grow to less manageable sizes. Second, the cost of operating on the data structures must be within a small factor of the cost of operating on the static graph, else these data structures will not be useful. Of these five data structures, three are approaches, one is semi-compact, but suited for fast operation, and one is focused on compactness and is a dynamic extension of any existing technique known as the WebGraph Framework. Our results show that for well intervalized graphs, like web graphs, the semi-compact is superior to all other data structures in terms of memory and access

time. Furthermore, we show that in terms of memory, the compact data structure outperforms all other data structures at the cost of a modest increase in update and access time.

We present a virtual memory subsystem which we implemented as part of the BDMPI runtime. Our new virtual memory subsystem, which we call SBMA, bypasses the operating system virtual memory manager to take advantage of BDMPI's node-level cooperative multi-taking. Benchmarking using a synthetic application shows that for the use cases relevant to BDMPI, the overhead incurred by the BDMPI-SBMA system is amortized such that it performs as fast as explicit data movement by the application developer. Furthermore, we tested SBMA with three different classes of applications and our results show that with no modification to the original program, speedups from $2\times$–$12\times$ over a standard BDMPI implementation can be achieved for the included applications.

We present a runtime system designed to be used alongside data parallel OpenMP programs for shared-memory problems requiring out-of-core execution. Our new runtime system, which we call OpenOOC, exploits the concurrency exposed by the OpenMP semantics to switch execution contexts during non-resident memory access to perform useful computation, instead of having the thread wait idle. Benchmarking using a synthetic application shows that modern operating systems support the necessary memory and execution context switching functionalities with high-enough performance that they can be used to effectively hide some of the overhead incurred when swapping data between memory and disk in out-of-core execution environments. Furthermore, we tested OpenOOC with practical computational application and our results show that with no structural modification to the original program, runtime can be reduced by an average of 21% compared with the out-of-core equivalent of the application.

# Contents

# List of Tables

# List of Figures

# List of Listings

# Chapter 1

# Introduction

A recent trend in many disciplines is the recognition of large scale data analysis as an integral component of the scientific discovery process. This new paradigm has been coined, "data-driven science" and is referred to by some as the "fourth paradigm" of science [2] — along with theoretical, experimental, and computational. This change is as much due to the increasing potential for scientists to generate and collect rich datasets, commonly dubbed the "data deluge", as it is to the robust computational tools being developed to process and extract knowledge from the data. Despite the richness of the data and the availability of sophisticated analysis tools, for many scientists, scientific inquiry is still limited. One major cause of this is the disproportionate increase in the size of datasets relative to the capacity of fast storage (main memory) in the computing systems where the data is analyzed. In other words, the datasets have outgrown the computers' ability to store the data in a place where it can be quickly accessed for analysis. This predicament can be referred to as *memory-constrained computation*. This is especially troublesome for the multitude of domain scientists who have access to large scale datasets, but do not have access to the class of computing system necessary to analyze the data in main memory. This thesis focuses on two methodologies to address the issue of memory-constrained computation, namely in-memory compression and external memory computation. It is explicitly divided into two parts, one for each methodology.

**In-memory compression** The first of these methodologies relies on our understanding of the hardware capabilities and limitations of modern computers. Namely, that within a computer, a memory hierarchy exists, such that accessing data that is stored at the top of the hierarchy (cache and main memory) is extremely fast, while accesses at the bottom (hard disk) are extremely slow. The catch is that the capacity of cache and main memory are orders of magnitude less than that of hard disk. Thus, for problems requiring datasets that are larger than the amount of main memory, a compact representation of the data, that will allow it to fit into main memory is required. If a main memory representation of the data can be realized, then the potential for improvements in analysis runtime is increased considerably. In this thesis, we explore two different approaches for creating compressed representations. One approach is to analyze the data as it is being generated, so called *in-situ* analysis. Based on the results of the analysis, the data can be reduced in way the preserves the ability to analyze later, but may not retain all components of the original data. Another approach is to preserve the original data as-is, but use standard compression techniques to identify redundancy in the data and then choose a representation for the data that exploits the redundancy.

**External-memory computation** An orthogonal methodology to solve the memory-constrained computation problem is external-memory computation, also known as out-of-core computation [3, 4]. External-memory computation is an algorithmic technique that exploits locality to reduce the amount of data movement to/from external memory [3, 4]. In this way, computations are localized to the partitions that reside in main memory as much as possible. Here external-memory means any type of secondary storage storage medium, whose access costs are significantly higher than main memory. Common Examples of this could be tape, spinning, or solid-state drives. This methodology addresses the inverse relationship between problem size and main memory size, but often means that algorithms need to be re-engineered to explicitly make considerations for the amount data movement required per computation. Further, for many external-memory computations, despite the explicit attention paid to minimizing data movement, the bandwidth between physical memory and secondary storage will still be be a major performance bottleneck.

As a result of these challenges, recent years have seen a significant amount of research

related to providing efficient and transparent external-memory computation [5–17]. However, current state-of-the-art solutions to this problem either introduce entirely new programming frameworks / languages that cooperate with associated runtimes to provide external-memory computation, or limit themselves to problems of some particular form. In this thesis, two external-memory solutions are proposed to this problem, each targeting a different memory architecture, which are extensions to existing and widely used programming frameworks.

## 1.1 Contributions

The contributions of this thesis are the development of effective and efficient solutions for solving computational problems on computing systems whose memory availability is insufficient for the desired problem. We show that under a variety of different circumstances, the algorithms and libraries discussed herein allow researchers to solve computational problems that would otherwise be inaccessible due to unfavorable memory limitations of available computing systems.

### 1.1.1 In-Memory Compression-Based Methods

**Scientific Data Compression** We developed a class of algorithms for compressing scientific simulation data computed on structured and unstructured grids. We propose a paradigm for lossy compression of this data in which the data computed on the grid is modeled as a graph, which gets decomposed into sets of vertices which satisfy a user defined error constraint $\epsilon$. Each set of vertices is replaced by a constant value with reconstruction error bounded by $\epsilon$. Our novel approach outperforms state-of-the-art lossy compression approaches and require on the average 25% of the space required by them for similar or better reconstruction quality. Moreover, the near linear complexity of our proposed algorithms makes them ideally suited for performing in situ compression in memory-constrained systems.

**Dynamic Graph Compression** We explore the space of data structures capable of representing dynamic sparse graphs in a format that is both compact and operationally efficient. We present existing and novel algorithms along with experimental analysis

for five data structures designed with this in mind. Of these five algorithms, three are baseline approaches, one is semi-compact, but suited for fast operation, and one is focused on compactness and is a dynamic extension of the state-of-the-art WebGraph Framework. Our results show that for well intervalized graphs, like web graphs, the semi-compact data structure is superior to all other data structures in terms of memory and access time. Furthermore, in the general case, the compact data structure outperforms all other data structures in terms of memory, at the cost of a modest increase in update and access time, making it ideal for graph operations in memory constrained systems.

### 1.1.2 External Memory-Based Methods

**Distributed Memory Architecture** We developed a virtual memory subsystem which we implemented as part of the Big Data Message Passing Interface library (BDMPI) runtime. Our new virtual memory subsystem, which we call SBMA, bypasses the operating system virtual memory manager to take advantage of BDMPI's node-level cooperative multi-taking. Benchmarking using a synthetic application shows that for the use cases relevant to BDMPI, the overhead incurred by the BDMPI-SBMA system is amortized such that it performs as fast as explicit data movement by the application developer. Furthermore, we tested BDMPI-SBMA with three different classes of applications and our results show that with no modification to the original program, speedups from 2×–12× over a standard BDMPI implementation can be achieved for the included applications.

**Shared Memory Architecture** We developed a runtime system to be used alongside data parallel OpenMP programs for shared-memory problems requiring out-of-core execution. Our new runtime system, which we call OpenOOC, exploits the concurrency exposed by the OpenMP semantics to switch execution contexts during non-resident memory access to perform useful computation, instead of having the thread wait idle. Benchmarking using a synthetic application shows that modern operating systems support the necessary memory and execution context switching functionalities with high-enough performance that they can be used to effectively hide some of the overhead incurred when swapping data between memory and disk in out-of-core execution environments. Furthermore, we tested OpenOOC with practical computational application

and our results show that with no structural modification to the original program, runtime can be reduced by an average of 21% compared with the out-of-core equivalent of the application.

## 1.2 Outline

This thesis is divided into two parts according to the type of approach used for solving the memory-constrained computing problem. The thesis is organized as follows:

- Part I includes Chapters 2 to 4, and focuses on compression based solutions.
  - In Chapter 2 an overview of prior work done on the subjects of scientific data compression and graph compression is presented along with necessary definitions and notations.
  - In Chapter 3, scientific data compression is addressed. A class of lossy compression algorithms is investigated, including detailed descriptions and analysis of the algorithms as well as experimental results using real-world datasets.
  - In Chapter 4, compressed data structures for representing graphs is addressed. More specifically, the space of data structures for representing dynamic sparse graphs is investigated in terms of computational efficiency for analysis queries, as well as memory efficiency. This is accompanied by an in-depth discussion of experimental results for a set of benchmark problems.
- Part II includes Chapters 5 to 7, and focuses on out-of-core computation based solutions.
  - In Chapter 5 an overview of prior work in the area of out-of-core computation is presented along with necessary definitions and notations.
  - In Chapter 6, the BigData Message Passing Interface (BDMPI) library and its virtual memory subsystem, called Storage Backed Memory Allocation (SBMA), are presented. The BDMPI-SBMA library addresses out-of-core computing in distributed memory systems. Along with a brief overview of the base BDMPI library, a detailed explanation of SBMA, is presented, Finally,

an experimental analysis is presented, comparing BDMPI-SBMA with the best known BDMPI configurations.

- In Chapter 7, the Open Out-of-core library (OpenOOC) library is presented. The OpenOOC library addresses out-of-core computing in shared memory systems. Along with a detailed description of the library, a detailed analysis of system characteristics is presented as well as an experimental analysis, comparing OpenOOC with the best known OS configurations.

- In Chapter 8, the conclusions are presented and several future research directions are proposed.

## 1.3 Related Publications

The work in this dissertation has been published in the following venues.

- **Jeremy Iverson**, Chandrika Kamath, and George Karypis. Fast and effective lossy compression algorithms for scientific datasets. In European Conference on Parallel Processing, 843–856, Springer, 2012.

- **Jeremy Iverson** and George Karypis. Storing dynamic graphs: speed vs. storage trade-offs. Technical Report 14-018, University of Minnesota, 2014.

- **Jeremy Iverson** and George Karypis. A memory management system optimized for bdmpi's memory and execution model. In Proceedings of the 22nd European MPI Users' Group Meeting, 2015.

- **Jeremy Iverson** and George Karypis. A virtual memory manager optimized for node-level cooperative multi-tasking in memory constrained systems. In The International Journal of High Performance Computing Applications, SAGE, 2017.

- **Jeremy Iverson** and George Karypis. OpenOOC: An OpenMP runtime companion for automating out-of-core execution in shared memory systems. Preparing for submission.

# Part I

# In-Memory Compression-Based Methods

# Chapter 2

# Preliminaries

## 2.1 Literature Reviews

### 2.1.1 Scientific Data Compression

The most popular techniques in this area are based on wavelet theory [18] that produces a compression-friendly sparse representation of the original data. To further sparsify this representation, coefficients with small magnitude are dropped with little impact on the reconstruction error [19, 20]. Depending on the intended use of the compressed representation, i.e., storage or visualization, these approaches may simply threshold the coefficients to reduce the amount of required data to store [19, 20], or they may create elaborate data structures [21, 22] to store the wavelet coefficients for progressive transmission or fast random access [23–25]. Due to the nature of the wavelet transform, classical wavelet methods apply only to structured grids. A notable application of this technique is the JPEG2000 image compression standard [26]. An alternative to wavelet compression is Adaptive Coarsening (AC) [27]. AC is an extension of the adaptive subsampling technique first introduced for transmitting HDTV signals [28], which is based on down-sampling a mesh in areas which can be reconstructed within some error tolerance and storing at full resolution the others. In [29], the authors use AC to compress data on structured grids and compare the results to wavelet methods. Even though AC can potentially be extended for unstructured grids [27], current implementations are limited to structured grids.

Another approach is spectral compression that extends the discrete cosine transform used in JPEG, from 2D regular grids to the space of any dimensional unstructured grids [30]. This method uses the Laplacian matrix of the grid to compute topology aware basis functions. The basis functions serve the same purpose as those in the wavelet methods and define a space where the data can be projected to, in order to obtain a sparse representation. Since the Laplacian matrix can be defined for the nodes of any grid, this method is not limited to structured grids. However, deriving the basis functions from the Laplacian matrix of large graphs is computationally prohibitive. For this reason, practical approaches first use a graph partitioning algorithm to decompose the underlying graph into small parts, and each partition is then compressed independently using spectral compression [30]. Finally, another approach, introduced in [31], is diffusion wavelets. The motivation for diffusion wavelets is the same as that of spectral compression, and is used to generate basis functions for a graph. However, instead of using the eigenvectors of the Laplacian matrix to derive these basis functions, diffusion wavelets generate them by taking powers of a diffusion operator. The advantage of diffusion wavelet is that its basis functions capture characteristics of the graph at multiple resolutions, while spectral basis functions only capture global characteristics.

### 2.1.2 Dynamic Graph Compression

One of the driving motivations for dynamic graph compression has been the alarming growth rate of the World Wide Web. A snapshot of the World Wide Web can be modeled as a graph, commonly referred to as the Web Graph, with Web pages represented as vertices and hyperlinks represented by directed edges. One of the first attempts at compressing the Web graph was the Link Database [32, 33]. The system achieved compression by representing each Web page with a numerical ID after lexicographically sorting the Web page URLs, then delta encoding the IDs of the neighbors of each vertex. This technique is known as gap encoding [33]. Another early attempt, by Adleret al. in [34], extended this technique by adding the notion of reference vertices. The authors developed an algorithm, which they called Find-Reference, that identified vertices in the graph that had many common neighbors. One of these vertices was chosen to be used as a reference when describing the neighbors of the other, thereby reducing the size of the second vertices neighbor list. Together these two works represent the basic

foundation about which most modern work on Web graph compression has been based.

Further attempts to compress the Web graph have largely relied on our deepening understanding of the structure of the Web graph itself. One of the most noteable such works is the WebGraph Framework [35, 36]. In their work, Boldi et al. leverage a great deal of study of the structure of the web [35] to introduce improved methods for finding references vertices and subsequently encoding neighbor lists. In [37], Boldi et al. investigated vertex reordering algorithms for the Web graph which are more amenable to compression. Later, the authors extended their work to social networks by considering alternative vertex reordering algorithms designed specifically for social networks [38].

Other work has been done to exploit the structure of the Web graph for the purpose of compression. In [39] the authors focused on distinguishing between the types of vertices in the neighbor list, local vs non-local, and then coding the lists accordingly. Alternatively, the authors of [40] represent the Web graph hierarchically, using so called S-node representations, to expose its locality structure, which is then exploited for efficient coding, much like previous work.

More recently, many attempts have been made to improve over the standard set by Boldi et al. [36]. The authors of [41] took an approach very similar to [39] and partition the set of neighbor connections into local and non-local. However, instead of using a single Huffman code to encode the connections, they opt for a more sophisticated encoding scheme, which results in compression rates slightly better than those of [36]. The authors of [42] took an different approach and relied on the well-known dictionary-based compression technique, Re-Pair [43], to compress the neighbor lists of vertices. This resulted in compression rates nearly identical to those of [36], but with much faster navigation of the graph. Lastly, another new and promising technique was introduced in [44], called Virtual Node Miner, which uses patten-mining to extract meaningful connectivity formations. By representing the common connectivity formations as "virtual nodes" in the compressed graph, compression is improved and certain computations can be performed directly on the compressed graph.

## 2.2 Definitions and Notations

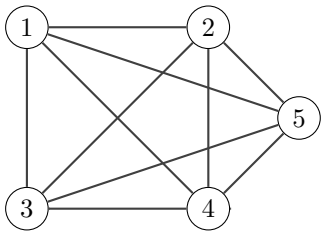
Many scientific simulations are carried out such that their underlying physical domain is modeled by a *grid*. A *grid* is a tessellation of Euclidean space by simple shapes such as triangles or tetraheda, often referred to as *cells*. The intersection of the lines that make up the cells are known as *nodes*. For the datasets explored in this thesis, the nodes are the "points-of-interest" in the space where some value(s) is being computed. Without loss of generality, we assume there is only one value being computed for each node of the grid. We also assume that the grid topology is fixed and thus can be represented and stored separately from the data which is computed on it.

A grid can be either structured or unstructured. A *structured grid* is a collection of cells which have an implicit geometric structure. That structure is a basic rectangular matrix structure, such that in $\mathbb{R}^3$, the nodes can be indexed by a triplet $(x, y, z)$. Thus, the grid topology can be described simply by the number of nodes in each of the three dimensions. An *unstructured grid* has no implicit structure. Since there is no implicit structure, the topology is described by identifying the cells which each node belongs to.

A graph $G = (V, E, L)$ consists of a set of vertices $V$, a set of edges $E$, and optionally a set of labels $L$. The set of adjacent vertices of a particular vertex $u$ is expressed as $\Gamma(u)$ and the degree of $u$ can then be stated as $\delta(u) = |\Gamma(u)|$. The maximum degree of any vertex in $G$ is denoted $\Delta(G) = \max_{\forall u \in V} \ \delta(u)$ and by abuse of notation, simply $\Delta$.

A *sparse graph* is a graph where $|V| \ll |E|$. There are two commonly used data structures for sparse graphs. The first is vertex-centric and is typically referred to as adjacency list representation. In this representation, each vertex is treated as an object and stores a list of the vertex ids of those vertices which it is adjacent to. In its most standard form, this representation uses linked-lists or arrays to store each vertex' adjacency list. However, the *compressed sparse row (CSR)* or *Yale format*, can also be used to store vertex adjacency lists in a more compact / cache friendly way. In this format, each vertex stores the index ($IA$) into an array ($JA$), which stores in consecutive indices, the ids of adjacent vertices for each vertex in the graph. Thus the edges which vertex $i$ is incident, can be identified as $JA[IA[i]]$ to $JA[IA[i+1]]$. The second is edge-centric and is best exemplified by the *coordinate list (COO)* format. In this format, each edge is represented as a pair of vertex ids $(u, v)$, where $u$ and $v$ are the endpoints

Figure 2.1: Example graph (left) along with its CSR format (middle) and COO format (right).




| CSR | |
|---|---|
| IA | JA |
| 0 | 2, 3, 4, 5 |
| 4 | 1, 3, 4, 5 |
| 8 | 1, 2, 4, 5 |
| 12 | 1, 2, 3, 5 |
| 16 | 1, 2, 3, 4 |
| 20 | |

| COO | |
|---|---|
| (1, 2) | (3, 4) |
| (1, 3) | (3, 5) |
| (1, 4) | (4, 1) |
| (1, 5) | (4, 2) |
| (2, 1) | (4, 3) |
| (2, 3) | (4, 5) |
| (2, 4) | (5, 1) |
| (2, 5) | (5, 2) |
| (3, 1) | (5, 3) |
| (3, 2) | (5, 4) |

of the edge. This means that the topology of the graph is expressed as a list of endpoint pairs. Each of these formats is illustrated in Figure 2.1. In both cases, the labels, $L$, can be stored along with or parallel to the edges.

A *grid graph* is an undirected graph that models the grid on which a scientific simulation is conducted. In a grid graph, the set of vertices, $V$, models the nodes of the grid for which values, $L$, are computed. The set of edges $E$, models the connectivity of adjacent nodes, where two nodes are adjacent if they are adjacent to the same cell in the grid.

A *web graph* is a directed graph whose vertex set $V$, represents pages from the World Wide Web and whose edge set $E$, captures the hyperlinks between the pages which exist in $V$. An *online social graph* is a directed or undirected graph where vertices represent individuals or organizations and the edges imply some type of relationship between the participating entities. Classic examples of online social graphs include Twitter (directed) and Facebook (undirected). In this thesis, all graphs are assumed to be undirected, since storage of directed graphs can be directly extended from storage of undirected graphs by storing an indicator of directedness with each edge or storing the graph transpose.

A *dynamic graph* is a graph whose vertex or edge sets are not static, i.e., they can

change. The vertex or edge sets of a dynamic graph may be changed as a consequence of *updates*: the addition and/or removal of vertices or edges, to the graph. In terms of graph data structures, *dynamic context* refers to the case in which the data structure used to store the graph is persisted across all updates.

An $\epsilon$-*bounded set-based decomposition* of $G$ is a partitioning of its set of vertices into non-overlapping sets $\{V_1, \ldots, V_k\}$ such that for each $V_i$, $\forall v_q, v_r \in V_i,\ |l_q - l_r| \leq \epsilon$ (i.e., each set contains vertices whose values differ at most by $\epsilon$). When the induced subgraph $R_i = (V_i, E_i)$ of $G$ is connected, the set $V_i$ will also be referred to as a *region* of $G$. When all sets in an $\epsilon$-bounded set-based decomposition form regions, then the decomposition will be referred to as an $\epsilon$-*bounded region-based decomposition* of $G$. Given a set of vertices $V_i$, the average value of its vertices will be referred to as its *mean value* and will be denoted by $\mu(V_i)$. Given a region $V_i$, its *boundary vertices* are its subset of vertices $B_i \subseteq V_i$ that are adjacent to at least one other vertex not in $V_i$, and its *interior vertices* are the subset of vertices $I_i \subseteq V_i$ that are adjacent only to vertices in $V_i$. Note that $I_i \cup B_i = V_i$.

# Chapter 3

# Scientific Data Compression

## 3.1 Introduction

Straightforward approaches for scientific data compression exist in lossless techniques designed specifically for floating-point data. However, due to the high variability of the representation of floating-point numbers at the hardware level, the compression factors realized by these schemes are often very modest [45,46]. Since most post-run analysis is robust in the presence of some degree of error, it is possible to employ lossy compression techniques rather than lossless, which are capable of achieving much higher compression rates at the cost of a small amount of reconstruction error. As a result, a number of approaches have been investigated for lossy compression of scientific simulation datasets including classical [18] and diffusion wavelets [31], spectral methods [30], and methods based on the techniques used for transmission of HDTV signals [28]. However, these approaches are either applicable only to simulations performed on structured grids or have high computational requirements for *in situ* data compression applications.

In this chapter we investigate the effectiveness of a class of lossy compression approaches that replace the actual values associated with sets of grid-nodes with a constant value whose difference from the actual value is bounded by a user-supplied error tolerance parameter. We develop approaches for obtaining these sets by considering only the nodes and their values and approaches that constrain these sets to connected subgraphs in order to further reduce the amount of information that needs to be stored. To ensure that these methods are applicable for *in situ* compression applications, our

work focuses on methods that have near-linear complexity and are equally applicable to structured and unstructured grids. We experimentally evaluate the performance of our approaches and compare it against that of other state-of-the-art data compression methods for scientific simulation datasets. Over our benchmark suite, our methods obtained compression of 1% of the original size with average PSNR of 43.00 and 3% of the original size with average PSNR of 63.30. Our experiments show that our methods achieve compressed representations, which on average, require 50%–75% less space than competing schemes at similar or lower reconstruction errors.

## 3.2 Methods

In this work we investigated the effectiveness of a lossy compression paradigm for grid-based scientific simulation datasets that replaces the values associated with a set of nodes with a constant value whose difference from the actual values is bounded. Specifically, given a graph $G = (V, E, L)$ modeling the underling grid, this paradigm computes an $\epsilon$-bounded set-based decomposition $\{V_1, \ldots, V_k\}$ of $G$ and replaces the values associated with all the nodes of each set $V_i$, with its mean value $\mu(V_i)$. This paradigm bounds the point-wise error to be no more than $\epsilon$, whose actual value is explicitly controlled by the users based on their subsequent analysis requirements. Since the values associated with the nodes tend to exhibit local smoothness [47], these value substitutions increase the degree of redundancy, which can potentially lead to better compression.

Following this paradigm, we developed two classes of approaches for obtaining the $\epsilon$-bounded set-based decomposition of $G$. The first class focuses entirely on the vertices of the grid and their values, where the second class also takes into account the connectivity of these vertices in the graph. In addition, we developed different approaches for encoding the information that needs to be stored on the disk in order to maximize the overall compression. The description of these algorithms is provided in the subsequent sections.

In developing these approaches, our research focused on algorithms whose underlying computational complexity is low because we are interested in being able to perform the compression *in-situ* with the execution of the scientific simulation on future exascale-class parallel systems. As a result of this design choice, the algorithms that we present

tend to find sub-optimal solutions but do so in time that in most cases is bounded by $O(|V| \log |V| + |E|)$.

### 3.2.1 Set-Based Decomposition

This class of methods derives the $\epsilon$-bounded set-based decomposition $\{V_1, \ldots, V_k\}$ of the vertices by focusing entirely on their values. Towards this end, we developed two different approaches. The first is designed to find the decomposition that has the smallest cardinality (i.e., minimize $k$), whereas the second is designed to find a decomposition that contains large-size sets.

The first approach, referred to as *SBD1*, operates as follows. The vertices of $G$ are sorted in non-decreasing order based on their values. Let $\langle v_{i_1}, \ldots, v_{i_n} \rangle$ be the sequence of the vertices according to this ordering, where $n$ is the number of vertices in $G$. The vertices are then scanned sequentially from $v_{i_1}$ up to vertex $v_{i_j}$ such that $l_{i_j} - l_{i_1} \leq \epsilon$ and $l_{i_{j+1}} - l_{i_1} > \epsilon$. The vertices in the set $\{v_{i_1}, \ldots, v_{i_j}\}$ satisfy the constraint of an $\epsilon$-bounded set and are used to form a set of the set-based decomposition. These vertices are then removed from the sorted sequence and the above procedure is repeated on the remaining part of the sequence until it becomes empty. It can be easily shown that the above greedy algorithm will produce a set-based decomposition that has the smallest number of sets for a given $\epsilon$.

The second approach, referred to as *SBD2*, utilizes the same sorted sequence of vertices $\langle v_{i_1}, \ldots, v_{i_n} \rangle$ but it uses a different greedy strategy for constructing the $\epsilon$-bounded sets. Specifically, it identifies the pair of vertices $v_{i_q}$ and $v_{i_r}$ such that $l_{i_r} - l_{i_q} \leq \epsilon$ and $r - q$ is maximized. The vertices in the set $\{v_{i_q}, \ldots, v_{i_r}\}$ satisfy the constraint of an $\epsilon$-bounded set and are used to form a set of the set-based decomposition. The original sequence is then partitioned into two parts: $\langle v_{i_1}, \ldots v_{i_{q-1}} \rangle$ and $\langle v_{i_{r+1}}, \ldots, v_{i_n} \rangle$, and the above procedure is repeated recursively on each of these subsequences. Note that the greedy decision in this approach is that of finding a set that has the most vertices (by maximizing $r - q$). It can be shown that SDB2 will lead to a decomposition whose maximum cardinality set will be at least as large as the maximum cardinality set of SBD1 and that the cardinality of the decomposition can be greater than that of SDB1's decomposition.

**Decomposition Encoding** We developed two approaches for encoding the vertex values derived from the $\epsilon$-bounded set-based decomposition. In both of these approaches, the encoded information is then further compressed using standard lossless compression methods such as GZIP [48], BZIP2 [49], and LZMA [50].

The first approach uses scalar quantization and utilizes a pair of arrays $Q$ and $M$. Array $Q$ is of size $k$ (the cardinality of the decomposition) and $Q[i]$ stores the mean value $\mu(V_i)$ of $V_i$. Array $M$ is of size $n$ (the number of vertices) and $M[j]$ stores the number of the set that vertex $v_j$ belongs to. During reconstruction, the value of $v_j$ is given by $Q[M[j]]$. Since for reasonable values of $\epsilon$, $k \ll n$, the number of distinct values in $M$ will be small, leading to a high degree of redundancy that can be exploited by the subsequent lossless compression step. We will refer to this approach as *scalar quantization encoding* and denote it by *SQE*.

The second approach encodes the information by sequentially storing the vertices that belong to each set of the decomposition. Specifically, it uses three arrays $Q$, $S$, and $P$, of sizes $k$, $k$, and $n$, respectively. Array $Q$ is identical to the $Q$ array of SQE and array $S$ stores the number of vertices in each set (i.e., $S[i] = |V_i|$). Array $P$ is used to store the vertices of each set in consecutive positions, starting with those of set $V_1$, followed by $V_2$, and so on. The vertices of each set are stored by first sorting them in increasing order based on their number and then representing them using a differential encoding scheme. The smallest numbered vertex of each set is stored as is and the number of each successive vertex is stored as the difference from the preceding vertex number. Since each vertex-set will likely have a large number of vertices, the differential encoding of the sorted vertex lists will tend to consist of many small values, and thus increase the amount of redundancy that can be exploited by the subsequent lossless compression step. We will refer to this approach as *differential encoding* and denote it by *DE*.

**Vertex Ordering** To achieve good compression using the above encoding schemes, vertices which are close in the vertex ordering should have similar values. Towards this end, we investigate three vertex orderings which are as follows. The first is the original ordering of the nodes, that is often derived by the grid generator and tends to have a spatial coherence. The second ordering is a breadth first traversal of the graph starting

from a randomly selected vertex. The third ordering is a priority first traversal, in which priority is given to those vertices which are adjacent to the most vertices which have been previously visited. Arranging the vertices according to their visit order is intended to put together in the ordering vertices that are close in the graph topology. Due to the local smoothness of values, vertices that appear close in the ordering will share similar values.

### 3.2.2 Region-Based Decomposition

This class of methods derives an $\epsilon$-bounded set-based decomposition $\{V_1, \ldots, V_k\}$ by requiring that each set $V_i$ also forms a region (i.e., its induced subgraph of $G$ is connected). The motivation behind this region-based decomposition is to reduce the amount of data that needs to be stored by only writing information about $V_i$'s boundary vertices and a select few of its interior vertices. During reconstruction, by taking advantage of $V_i$'s connectivity, its non-saved interior vertices can be identified by a depth- or breadth-first traversal of $G$ starting at the saved interior vertices and terminating at its boundary vertices. The set of vertices visited in the course of this traversal will be exactly those in $V_i$. From this discussion, we see that the amount of compression that can be achieved by this class of methods is directly impacted by the number of boundary vertices that must be stored. Thus, the region identification approaches must try to reduce the number of boundary vertices. Towards this end, we developed three different heuristic approaches whose description follows.

The first approach, referred to as *RBD1*, is designed to compute a decomposition that minimizes the number of regions. The motivation behind this approach is that by increasing the average size of each region (due to a reduction in the decomposition's cardinality), the number of interior vertices will also increase. RBD1 initially sorts the vertices in a way identical to SBD1, leading to the sorted sequence $s = \langle v_{i_1}, \ldots, v_{i_n} \rangle$. Then, it selects the first vertex in the sequence ($v_{i_1}$), assigns it to the first region $V_1$, and removes it from $s$. It then proceeds to select from $s$ a vertex $v_{i_j}$ that is adjacent to at least one vertex in $V_1$ and $l_{v_{i_j}} - l_{v_1} \leq \epsilon$, inserts it into $V_1$, and removes it from $s$. This step is repeated until no such vertex can be selected or $s$ becomes empty. The above algorithm ensures that $V_1$ is an $\epsilon$-bounded set and that the subgraph of $G$ induced by $V_1$ is connected. Thus, $V_1$ is a region and is included in the region-based

decomposition. The above procedure is then repeated on the vertices remaining in $s$, each time identifying an additional region that is included in the decomposition. Note that unlike the algorithm for SBD1, the above algorithm does not guarantee that it will identify the $\epsilon$-bounded region-based decomposition that has the minimum number of regions.

The second approach, referred to as *RBD2*, is designed to compute a decomposition that contains large regions, as the regions that contain a large number of vertices will also tend to contain many interior vertices. One way of developing such an algorithm is to use the greedy approach similar to that employed by SBD2 to repeatedly find the largest region from the unassigned vertices and include it in the decomposition. However, due to the region's connectivity requirement, this is computationally prohibitive. For this reason, we developed an algorithm that consists of two steps. The first step is to obtain an $\epsilon$-bounded set-based decomposition $\{V_1, \ldots, V_k\}$ using SBD1. The second step is to compute an $\epsilon$-bounded region-based decomposition of each set $V_i$. The union of these regions over $V_1, \ldots, V_k$ is then used as the region-based decomposition computed by RBD2. This two-step approach is motivated by the following observation. One of the reasons that prevents RBD1 from identifying large regions is that it starts growing each successive region from the lowest-valued unassigned vertex and does not stop until all of the unassigned vertices adjacent to that region have values that will violate the $\epsilon$ bound. This will tend to fragment subsequent regions as the are constrained by the initial vertices that have low values. RBD2, by forcing RBD1's region identification algorithm to stay within each set $V_i$, prevents this from happening and as our experiments will later show, lead to a decomposition that has smaller number of boundary vertices and better compression.

Finally, the third approach, referred to as *RBD3*, is designed to directly compute a decomposition whose regions have a large number of interior vertices. It consists of three distinct phases. The first phase identifies a set of *core* regions that contain at least one interior vertex, the second phase expands these regions by including additional vertices to them, and the third phase creates non-core regions. Let $V'$ be the subset of vertices of $V$ such that $\forall v \in V', v \cup \text{adj}(v)$ is an $\epsilon$-bounded set, where $\text{adj}(v)$ is the set of vertices adjacent to $v$. A core region, $V_i$, is created as follows. An unassigned vertex $v \in V'$ whose adjacent vertices are also unassigned is randomly selected and $v \cup \text{adj}(v)$

is inserted into $V_i$. Then the algorithm proceeds to identify an unassigned vertex $u \in V'$ such that: (i) it is connected to at least one vertex in $V_i$, (ii) all the vertices in $\text{adj}(u) \setminus V_i$ are also unassigned, and (iii) $V_i \cup \{u\} \cup \text{adj}(u)$ is an $\epsilon$-bounded set. If such a vertex $u$ exists, then $u$ and $\text{adj}(u) \setminus V_i$ are inserted into $V_i$. If no such vertex exists, then $V_i$'s expansion stops. The above procedure is repeated until no more core regions can be created. Note that by including $u$ and its $\text{adj}(u) \setminus V_i$ vertices into $V_i$, we ensure that $u$ becomes an interior vertex of $V_i$. During the second phase of the algorithm, the vertices that have not been assigned to any region are considered. If a vertex $v$ can be included to an existing region while the resulting region remains an $\epsilon$-bounded set, then it is assigned to that. Finally, the third phase is used to create additional regions containing the remaining unassigned vertices (if they exist), which is done using RBD1.

**Decomposition Encoding** As discussed earlier, the region-based decomposition allows us to reduce the storage requirements by storing only the boundary vertices along with the interior vertices that are used as the *seeds* of the (depth- or breadth-first) traversals. For each region $V_i$, the set of seed-vertices $I_i^s$ is determined as follows. An interior vertex is randomly selected, added to $I_i^s$, and a traversal from that vertex is performed terminating at $V_i$'s boundary vertices. If any of $V_i$'s interior vertices has not been visited, then the above procedure is repeated on the unvisited vertices, each time adding an additional source vertex into $I_i^s$. In most cases, one seed vertex will be sufficient to traverse all the interior vertices, but when regions are contained within other regions, multiple seed vertices may be required. Also, in the cases in which $V_i$ consists of only boundary vertices, $I_i^s$ will be empty.

An additional storage optimization is possible, as there is no need to store the boundary vertices for all the regions. In particular, consider a region $V_i$ and let $\{V_{i_1}, \ldots, V_{i_m}\}$ be the set of its adjacent regions in the graph. We can then identify $V_i$ by performing a traversal from the vertices in $I_i^s$ that terminates at the boundary vertices of $V_i$'s adjacent regions. All the vertices visited during that traversal (excluding the boundary vertices) along with $I_i^s$ will be exactly the vertices of $V_i$. Thus, we can choose not to store $V_i$'s boundary vertices as long as we store the boundary vertices for all of its adjacent regions. In our algorithm, we choose the regions whose boundary information will not be stored in a greedy fashion based on the size of their boundaries. Specifically, we

construct the region-to-region adjacency graph (i.e., two regions are connected if they contain vertices that are adjacent to each other), assign a weight to the vertex corresponding to $V_i$ that is equal to $|B_i|$ (i.e., the size of its boundary), and then identify the regions whose boundary information will not be stored by finding a maximal weight independent set of vertices in this graph using a greedy algorithm.

Given the above, we can now precisely describe how the region-based decomposition is stored. Let $\{V_1, \ldots, V_k\}$ be the $\epsilon$-bounded region-based decomposition, $B_1, \ldots, B_k$ be the sets of boundary vertices that need to be stored (if no boundary information is stored for a region due to the earlier optimization, then the corresponding boundary set is empty), and $I_1^s, \ldots, I_k^s$ be the sets of internal seed-vertices that have been identified. Our method stores five arrays, $Q$, $N_I$, $N_B$, $I_I$, and $I_B$. The first three arrays are of length $k$, $I_I$ is of length equal to the total number of seed vertices ($\sum_i |I_i^s|$), and $I_B$ is of length equal to the total number of boundary vertices ($\sum_i |B_i|$). Array $Q$ stores the mean values of each region, whereas arrays $N_I$ and $N_B$ store the number of seed and boundary vertices of each region, respectively. Array $I_I$ stores the indices of the regions in consecutive order starting from $I_1^s$, whereas array $I_B$ is used to store the boundary vertices of each region in consecutive positions starting from $B_1$. These indices are stored using the same differential encoding approach described in Section 3.2.1 and like that approach, the results of this encoding are further compressed using a standard lossless compression method.

## 3.3 Experimental Design

**Datasets** We evaluated our algorithms using seven real world datasets obtained from researchers at UMN and our colleagues at NASA and LLNL. These datasets correspond to fluid turbulence and combustion simulations and contain both structured and unstructured grids. Their characteristics are shown in Table 3.1.

**Evaluation Methodology & Metrics** We measured the performance of the various approaches along two dimensions. The first is the error introduced by the lossy compression and the second is the degree of compression that was achieved. The error was measured using three different metrics: (i) the root mean squared error (RMSE), (ii)

Table 3.1: Information about the various datasets.

| Dataset | $\|V\|$ | $\|E\|$ | $\mu(V)$ | Grid Type |
|---|---|---|---|---|
| d1 | 486051 | 4335611 | 0.9958 | unstruct. |
| d2 | 589824 | 1744896 | 0.5430 | struct. |
| d3 | 1936470 | 15399496 | 0.9874 | unstruct. |
| d4 | 16777216 | 50102272 | 163.70 | struct. |
| d5 | 31590144 | 94562224 | 0.0176 | unstruct. |
| d6 | 41472000 | 123926400 | 0.2107 | struct. |
| d7 | 100663296 | 300744704 | 4.5644 | struct. |

the maximum point-wise error (MPE), and (iii) the peak signal-to-noise ratio (PSNR). The RMSE is defined as

$$RMSE = \sqrt{\frac{1}{|V|}\sum_{i=1}^{|V|} |l_j - \hat{l}_j|^2}, \tag{3.1}$$

where $l_j$ is the original value of vertex $v_j$ and $\hat{l}_j$, is its reconstructed value. RMSE is a standard measure of difference between observed values and their approximations. The MPE is defined as

$$MPE = \max(|l_1 - \hat{l}_1|, ..., |l_n - \hat{l}_n|), \tag{3.2}$$

which is the $\ell_\infty$-norm of the point-wise error vector. The MPE measure is presented in tandem with RMSE to identify those algorithms which achieve low RMSE, but sustain high point-wise errors. Finally, the PSNR is defined as

$$PSNR = 20 \cdot \log_{10}\left(\frac{\max(x_1, ..., x_n)}{RMSE}\right), \tag{3.3}$$

which is a normalized error measure; thus, facilitating comparisons of error between datasets with values that differ greatly in magnitude. The compression effectiveness was measured by computing the compression ratio (CR) of each method, which is defined as follows:

$$CR = \frac{\text{compressed size}}{\text{uncompressed size}}. \tag{3.4}$$

The wavelet and spectral methods were implemented in Matlab®. The spectral method uses METIS [51] as a pre-processing step to partition the graph before compressing. The adaptive coarsening implementation was acquired from the authors of [29] and modified to provide the statistics necessary for these experiments. All algorithms described in Section 3.2 were implemented in C++. Finally, for the lossless compression of the decomposition encodings, we used LZMA compression (7-zip's implementation) as it resulted in better compression than either GZIP or BZIP2. In addition, the same LZMA-based compression was applied to the output of the spectral and wavelet-based compressions. Note that AC does not need that because it achieves its compression by coarsening the graph and reducing the data output.

## 3.4 Results

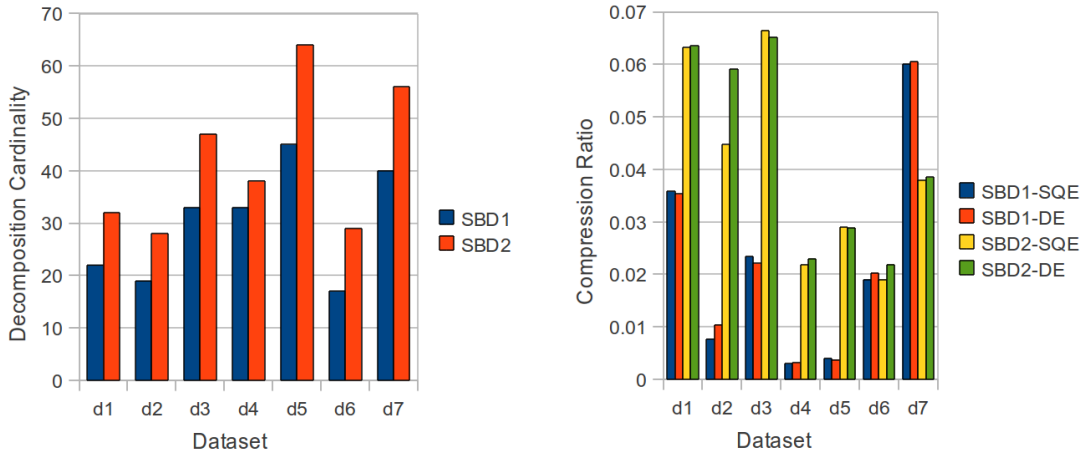
Our experimental evaluation is done in two parts. First, we select a fixed set of values for RMSE and compare the various algorithmic choices for the set- and region-based decomposition approaches in terms of their compression ability. Second, we compare the compression performance of the best combinations of these schemes against that achieved by other approaches for two different levels of lossy compression errors.

### 3.4.1 Set-Based Decomposition

Figure 3.1 shows the compression performance achieved by SBD1 and SBD2 for the different datasets across the different decomposition encoding schemes described in Section 3.2.1. These results show that SBD1 tends to perform better than SBD2 and on average, it requires 5% less storage for each specific combination of decomposition encoding and vertex ordering scheme. This can be attributed to the fact that the cardinality of its decomposition is often considerably lower than SBD2's, which tends to outweigh the benefits achieved by the few larger sets identified by SBD2.

Comparing the performance of the decomposition encoding schemes (SQE and DE), we see that SQE performs considerably better across both decomposition methods and ordering schemes. On the average, SQE requires only 75% of the storage of DE. These results suggest that when compared to scalar quantization, the differential encoding of the vertices in each set is not as effective in introducing redundancy in the encoding,

Figure 3.1: Statistics for set-based decomposition.

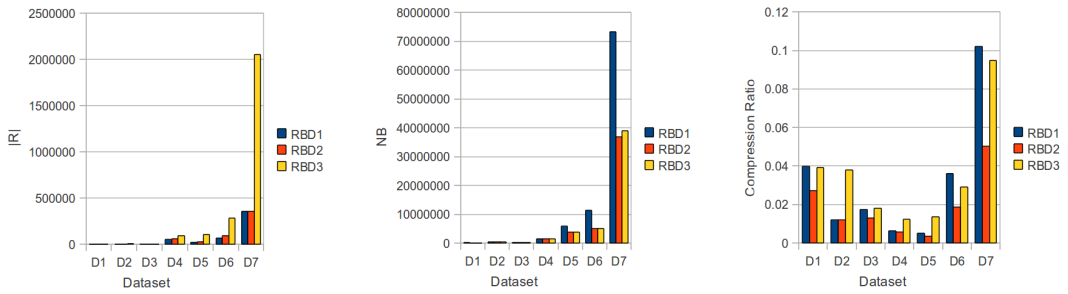



which in turn reduces the compression that can be obtained by the lossless LZMA compression. Finally, comparing the performance of the three vertex ordering schemes, we found that the original ordering leads to greater compression than either of the breadth first traversal or the priority first traversal. As discussed in Section 3.2.1, this ordering utilizes information from the underlying grid geometry, and as such it has a higher degree of regularity, leading to better compression. With respect to the other two methods, we found that the priority first traversal tends to perform better than breadth first.

### 3.4.2 Region-Based Decomposition

Figure 3.2 shows various statistics of the decompositions computed by RBD1, RBD2, and RBD3 for the different datasets and their compression performance for the three vertex ordering schemes. In terms of the number of regions into which $G$ is decomposed, we see that RBD1 results in the least number of regions, whereas RBD3 identifies a considerably greater number of regions (often 2–7 times more regions than RBD1). We also see that RBD2 only identifies slightly more regions than RBD1 (about 18% more on average). In terms of the number of boundary vertices that need to be stored by each decomposition, we see an inversion of the previous results. RBD2 and RBD3 produce the smallest boundary sets, typically being within about 5% of each other, whereas RBD1

Figure 3.2: Statistics for region-based decomposition, $|R|$ refers to number of regions identified, and NB refers to number of boundary vertices after storage optimization described in Section 3.2.1.




produces boundary sets which are considerably larger, in some cases, more than twice the size of those required by RBD2 and RBD3. These results suggest that the region identification heuristics employed by RBD2 and RBD3 are quite effective in minimizing the total number of boundary vertices, even though they find more regions.

In terms of compression performance, we see that across all datasets RBD2 results in the lowest compression ratio. On the average, RBD2 requires only 70% of the storage of RBD1 and 56% of RBD3. Contrasting this with the number of boundary vertices identified by each approach, we see that there is a direct correlation, based on the size of the boundary vertex set, between RBD1 and RBD2 in terms of which approach results in lower compression ratio and by how much. RBD3 does not share in this correlation, due to its significantly higher number of regions.

### 3.4.3 Comparison with Other Methods

In our last set of experiments, we compare the performance of the best-performing combinations of the set- and region-based decomposition approaches (SBD1 with SQE encoding and original vertex ordering, and RBD2 with original vertex ordering) against wavelet compression (Wvlt), spectral compression (Spctrl), and adaptive coarsening (AC). Among these techniques, the wavelet compression and adaptive coarsening can only be applied to structured grids and are only presented for the d2, d4, d6, and d7 datasets. Also, due to its high computational requirements, we were not able to obtain

results for the spectral compression for the largest problem (d7). In addition to these schemes, we also experimented with diffusion wavelets [31]. However, we obtained poor compression and we omitted those results.

Tables 3.2 and 3.3 shows the results of these experiments for two different compression levels, labeled "high error tolerance" and "low error tolerance". These compression levels result in RMSEs and MPEs that differ by approximately an order of magnitude, and were obtained by experimenting with the parameters of the various schemes so that to match their RMSEs for each of the datasets. However, for AC we were unable to achieve the desired RMSEs at all error tolerance levels. In the case that we could not achieve a desired RMSE, the results were omitted.

The results show that on average, our algorithms compress the simulation datasets to 2–5% of their original size. Compared with just lossless compression only, which results in storage costs of 40–80% of the original size, this is a big improvement. The results also show that for all but two experiments, SBD1 performs the best and that on average it required only 36% of the storage of the next best algorithm. For unstructured grids it requires on average 25% of the storage of Spctrl whereas for structured grids it requires on average 48% and 38% of the space of Wvlt and AC, respectively. Moreover, we see that as the amount of allowable error is lowered, the performance gap between SBD1 and the other methods grows. In addition, for unstructured grids, RBD2 performs the second best overall and requiring 61% of the space required by the Spctrl on average. We also see that due to the $\epsilon$ constraint placed on the our methods, they consistently result in MPE values which are much lower than those of the competing algorithms. These results suggest that in the context of grid-based simulation, SBD1 and RBD2 are consistently good choices for compression, providing low point-wise and global reconstruction error, high compression ratio, and low computational complexity.

Table 3.2: Comparison of scientific data compression algorithms for high error tolerance.

| Dataset | Algorithm | RMSE | PSNR | MPE | CR |
|---|---|---|---|---|---|
| | SBD1 | 6.30E-03 | 4.64E+01 | 1.89E-02 | 2.39E-02 |
| d1 | RBD2 | 6.28E-03 | 4.65E+01 | 1.89E-02 | **2.52E-02** |
| | Spctrl | 6.37E-03 | 4.63E+01 | 1.11E-01 | 4.00E-02 |
| | SBD1 | 2.92E-02 | 3.60E+01 | 7.33E-02 | **2.51E-03** |
| | RBD2 | 2.88E-02 | 3.61E+01 | 8.02E-02 | 5.02E-03 |
| d2 | Wvlt | 3.10E-02 | 3.55E+01 | 2.34E-01 | 2.00E-02 |
| | Spctrl | 3.17E-02 | 3.53E+01 | 7.34E-01 | 4.50E-02 |
| | AC | 3.31E-02 | 3.49E+01 | 1.50E-01 | 1.86E-02 |
| | SBD1 | 5.22E-03 | 4.88E+01 | 1.91E-02 | **1.27E-02** |
| d3 | RBD2 | 5.18E-03 | 4.89E+01 | 1.93E-02 | 1.33E-02 |
| | Spctrl | 5.27E-03 | 4.87E+01 | 2.14E-01 | 4.50E-02 |
| | SBD1 | 2.36E+01 | 4.70E+01 | 1.63E+02 | **2.63E-03** |
| | RBD2 | 2.05E+01 | 4.83E+01 | 1.65E+02 | 6.30E-03 |
| d4 | Wvlt | 2.47E+01 | 4.66E+01 | 6.86E+02 | 7.50E-03 |
| | Spctrl | 2.57E+01 | 4.63E+01 | 1.78E+03 | 3.50E-02 |
| | AC | 2.30E+01 | 4.73E+01 | 3.01E+03 | 2.15E-02 |
| | SBD1 | 4.97E-04 | 4.59E+01 | 1.78E-03 | **3.22E-03** |
| d5 | RBD2 | 4.88E-04 | 4.61E+01 | 1.76E-03 | 4.47E-03 |
| | Spctrl | 5.84E-04 | 4.45E+01 | 4.56E-02 | 5.00E-03 |
| | SBD1 | 1.21E-02 | 3.82E+01 | 5.70E-02 | 9.30E-03 |
| | RBD2 | 1.20E-02 | 3.82E+01 | 5.71E-02 | 1.28E-02 |
| d6 | Wvlt | 9.48E-03 | 4.03E+01 | 1.56E-01 | **5.00E-03** |
| | Spctrl | 1.60E-02 | 3.57E+01 | 6.37E-01 | **5.00E-03** |
| | AC | 1.82E-02 | 3.46E+01 | 1.50E-01 | 1.11E-02 |
| | SBD1 | 2.72E-01 | 4.27E+01 | 5.50E-01 | 2.82E-02 |
| | RBD2 | 2.70E-01 | 4.28E+01 | 7.41E-01 | 3.43E-02 |
| d7 | Wvlt | 2.76E-01 | 4.26E+01 | 2.75E+00 | **1.00E-02** |
| | AC | 2.76E-01 | 4.26E+01 | 1.00E+00 | 1.82E-02 |

Table 3.3: Comparison of scientific data compression algorithms for low error tolerance.

| Dataset | Algorithm | RMSE | PSNR | MPE | CR |
|---|---|---|---|---|---|
| | SBD1 | 6.66E-04 | 6.60E+01 | 1.86E-03 | **7.80E-02** |
| d1 | RBD2 | 6.38E-04 | 6.63E+01 | 2.12E-03 | 1.28E-01 |
| | Spctrl | 3.90E-03 | 5.06E+01 | 7.14E-02 | 1.05E-01 |
| | SBD1 | 2.50E-03 | 5.74E+01 | 7.71E-03 | **1.27E-02** |
| | RBD2 | 1.91E-03 | 5.97E+01 | 7.71E-03 | 6.57E-02 |
| d2 | Wvlt | 2.59E-03 | 5.70E+01 | 2.38E-02 | 1.15E-01 |
| | Spctrl | 7.04E-03 | 4.84E+01 | 3.56E-01 | 1.30E-01 |
| | AC | 6.80E-03 | 4.87E+01 | 7.19E-01 | 5.17E-02 |
| | SBD1 | 4.79E-04 | 6.96E+01 | 2.05E-03 | **3.56E-02** |
| d3 | RBD2 | 4.54E-04 | 7.00E+01 | 2.07E-03 | 4.33E-02 |
| | Spctrl | 3.31E-03 | 5.28E+01 | 1.35E-01 | 1.00E-01 |
| | SBD1 | 2.43E+00 | 6.68E+01 | 1.34E+01 | **1.02E-02** |
| | RBD2 | 2.00E+00 | 6.85E+01 | 1.36E+01 | 3.51E-02 |
| d4 | Wvlt | 2.64E+00 | 6.61E+01 | 4.87E+01 | 2.50E-02 |
| | Spctrl | 3.92E+00 | 6.26E+01 | 3.59E+02 | 1.95E-01 |
| | AC | - | - | - | - |
| | SBD1 | 5.43E-05 | 6.51E+01 | 1.28E-04 | **1.42E-02** |
| d5 | RBD2 | 5.32E-05 | 6.53E+01 | 1.69E-04 | 4.96E-02 |
| | Spctrl | 5.87E-05 | 6.45E+01 | 8.74E-03 | 6.50E-02 |
| | SBD1 | 1.05E-03 | 5.94E+01 | 4.87E-03 | **2.96E-02** |
| | RBD2 | 8.75E-04 | 6.10E+01 | 4.87E-03 | 1.74E-01 |
| d6 | Wvlt | 1.05E-03 | 5.94E+01 | 1.17E-02 | 5.50E-02 |
| | Spctrl | 1.05E-03 | 5.94E+01 | 4.32E-02 | 6.50E-02 |
| | AC | - | - | - | - |
| | SBD1 | 2.74E-02 | 6.26E+01 | 6.37E-02 | **8.70E-02** |
| | RBD2 | 2.17E-02 | 6.47E+01 | 7.99E-02 | 5.16E-01 |
| d7 | Wvlt | 3.05E-02 | 6.17E+01 | 2.00E-01 | 1.60E-01 |
| | AC | - | - | - | - |

# Chapter 4

# Dynamic Graph Compression

## 4.1 Introduction

With the staggering growth rate of web graphs and online social network datasets in the past decade, and the increasing demand for more efficient query and analysis of these networks, comes a renewed interest in the data structures and algorithms with which we represent and explore these networks. Historically, researchers have been limited to studying static snapshots of the networks off-line. However, this simplistic approach has two major drawbacks. First, real-time analysis is not possible when the network in question must be captured and output before analysis can commence. Second, the exploration of static networks is inherently limited by their time independent nature. It has been shown that the evolution of networks can illuminate interesting and important insights about the structure of the network [52].

Recently, researchers have acknowledged these limitations of static network analysis and much of the focus has shifted to dynamic network analysis. However, many of the contributions which can be attributed to this shift are aimed at specific problem instances. For example, there is a great deal of literature related to the data structures and algorithms necessary for maintaining network connectivity [53], shortest paths [54], or maximal cliques [55], but there is a dearth of research addressing the problem of which data structures are most appropriate for a general dynamic network framework on top of which more complex analysis can be performed.

As graph analysis techniques continue to evolve, the desire to perform this analysis

in a dynamic context is ever increasing. In this work we investigate data structures for dynamic sparse graphs, with special attention being paid to memory requirements and access times. Specifically, given a graph $G = (V, E)$, we want to represent $G$ using as little memory as possible and at the same time, achieve access times which are as low as possible. Due to the nature of sparse graphs, $|V| \ll |E|$, edge updates are significantly more likely than vertex updates. For this reason, edge updates will be the focus of this thesis. However, an algorithm for vertex updates is presented and is included in the experimental results. Furthermore, in each of the data structures described in this work, an algorithm for edge removal can easily be deduced from the algorithm for edge insertion. Thus, due to lack of space, we will not go into detail on edge removal algorithms.

In this chapter we investigate the space of data structures for representing dynamic networks in terms of computational efficiency for network analysis queries, as well as memory efficiency. The latter is an important factor, since the dynamic networks which are the focus of this thesis are web and online social networks, both of which are increasingly large. Along with previously known dynamic network data structures like: Linked-List (LL), Batch Compressed Sparse Row (BCSR), and Dynamic Adjacency Array (DAA), we introduce two extensions of static network representations: Dynamic Intervalized Adjacency Array (DIAA) and Dynamic Compressed Adjacency Array (DCAA), which are especially applicable to the types of networks addressed by this thesis. We theoretically and experimentally evaluate each of the data structures in terms of operation time and memory requirements. Over our benchmark suite, we show that in terms of operation time, BCSR is superior to the other data structures and LL is inferior in all cases. We also show that BCSR requires more memory under all circumstances and does not scale as well as DIAA. In terms of memory, DCAA outperforms all other data structures, but in most cases, does so at the cost of operation time. DAA and DIAA strike a balance between memory requirements and operation time, with DIAA exceeding DAA in operation time, as well as memory, for well intervalized graphs, such as web graphs.

Table 4.1: Storage and update complexity.

| Data structure | Storage | Edge Update |
|---|---|---|
| LL | $\Theta(\|V\| + 2\|E\|)$ | $O(\Delta)$ |
| BCSR | $O(\|V\| + 2\|E\|)$ | $O(\Delta)$ |
| DAA | $\Theta(\|V\| + \|E\|)$ | $O(\Delta)$ |
| DIAA | $O(\|V\| + \|E\|)$ | $O(\Delta)$ |
| DCAA | $O(\|V\| + \|E\|)$ | $O(3\Delta)$ |

* BCSR also incurs a cost to fold LL into static CSR, which is $O(|E|)$ over all folds

## 4.2 Methods

We investigate five data structures for representing dynamic sparse graphs. In each approach, a different trade-off is made to improve memory requirements or access time. Asymptotically, each of the five approaches exhibits identical behavior, namely, $O(|V| + |E|)$ memory and $O(\Delta)$ time for edge update; however, in practice, memory access patterns and internal data structure representation can cause significant variations in absolute memory used or access time. The $O(\Delta)$ update complexity is a direct result of the requirement that $\forall u \in V$, $\Gamma(u)$ is stored in sorted order. Maintaining this invariant is typical of many applications, due to its requirement for many graph operations, i.e., finding common neighbors of a vertex (set intersection). A sixth approach, based on the packed-memory array data structure described in [56] and applied to dynamic graphs in [57] was also implemented and experimented. However, due to the complex maintenance procedure required for edge updates, its runtime was prohibitively high and will not be discussed further.

### 4.2.1 Linked-List (LL)

The first data structure, referred to as *Linked-List (LL)*, is based on the standard singly-linked list representation of a sparse graph adjacency list. In LL, each vertex is an object which has a linked-lists of nodes. Each node in the linked-list contains the vertex of a single adjacent vertex. The memory required for this data structures is $\Theta(|V| + 2|E|)$

and the cost to update an edge is $\Delta$, since the adjacency list needs to be maintained in sorted order [58].

### 4.2.2 Batch Compressed Sparse Row (BCSR)

There are two main drawbacks with LL. The first is that inserting into a linked-list while maintaining some ordering gets more expensive as the size of the data structure grows. Second, due to the pointer-based nature of linked-lists, the memory access patterns of linked-lists tend not to be cache friendly, making access to consecutive nodes of a linked-list more costly than access to consecutive elements of an array.

In BCSR the graph is represented as a combination of two data structures: a static CSR and a LL. Each time that a new edge is added, it is added directly into the LL. Again, the linked-list for vertex $u$ and $v$ are updated for edge $(u, v)$. When the LL has grown significantly large, so that it is no longer memory or access efficient, it is folded into the static CSR. This means that the static CSR is appropriately resized to hold all edges from the graph. Then the static CSR is rebuilt to include the edges which it contained previously, as well as the edges from the LL. As edges are included in the static CSR, they are removed from the LL. Upon completion of a fold routine, the static CSR contains all edges currently in the graph and the LL is empty.

This type of batch processing approach has two benefits. First, by periodically reducing the size of the linked-list, the insertion time will be reduced. Second, the majority of the graph is stored in a data structure which has better properties for data access than the linked-list. Since BCSR is built upon both a static CSR representation as well as a LL, the memory requirement will be $\Omega(|V|+|E|)$ and $O(|V|+2|E|)$, depending on the interval at which the LL is folded into the static CSR. The trade-off in this type of approach is determining the appropriate time to fold the LL into the static CSR. Many policies can be designed to govern this decision, including, but not limited to, simple schemes such as folding after a constant number of updates have occurred, to more complex schemes based on timing data structure operations and folding when it becomes more expensive to operate on the unfolded data structure than to fold it and then operate.

Like the memory requirements for BCSR, the cost of updating an edge will depend on the interval at which the data structure is folded and will be $\Omega(1)$ and $O(\Delta)$. However,

Figure 4.1: Adjacency array information for a subset of vertices from an example graph.

| Vertex | Degree | Adjacency array |
|---|---|---|
| ... | ... | ... |
| 13 | 8 | 1, 2, 3, 8, 9, 10, 11, 14 |
| 14 | 4 | 13, 15, 17, 42 |
| 15 | 5 | 14, 32, 33, 34, 35 |
| ... | ... | ... |
| 32 | 8 | 2, 3, 4, 5, 10, 11, 12, 15 |
| 33 | 6 | 1, 2, 3, 5, 15, 23 |
| 34 | 5 | 1, 2, 3, 5, 6, 8 |
| ... | ... | ... |

along with the cost of edge updates comes the cost of incorporating the LL into the static CSR. This requires iterating all edges currently in the graph. As long as the size of the static CSR is expanded by some constant proportion, it can be shown that the amortized cost of folding the LL into the static CSR carries a cost of $O(|V| + |E|)$ over all folds [58].

### 4.2.3 Dynamic Adjacency Array (DAA)

Implementing a data structure based on a linked-list means that there will always be a 2x memory overhead, since each linked-list node must store a value as well as a pointer to the next node. In the third implementation, called *Dynamic Adjacency Array (DAA)*, this memory overhead is eliminated by assigning to each vertex a dynamically allocated array in which to stores the ids of adjacent vertices. The size of this dynamically allocated array is equal to exactly the current degree of the vertex. In order to iterate or search the adjacency array of a particular vertex it is necessary to know its current degree. Thus, this value is also stored and maintained across updates as part of the vertex object, along with the adjacency array. The storage of degree along with adjacency arrays in each vertex object make the memory requirement of this data structure $\Theta(|V| + |E|)$.

Figure 4.2: Intervalized adjacency array for example graph from Figure 4.1.

| Vertex | Degree | Intervals | Residuals |
|---|---|---|---|
| ... | ... | ... | ... |
| 13 | 8 | [1, 3], [8, 4] | 14, |
| 14 | 4 | | 13, 15, 17, 42 |
| 15 | 5 | [32, 4] | 14 |
| ... | ... | ... | ... |
| 32 | 8 | [2, 4], [10, 3] | 15 |
| 33 | 6 | [1, 3] | 5, 15, 23 |
| 34 | 6 | [1, 3], [5, 2] | 8 |
| ... | ... | ... | ... |

When a new edge $(u, v)$ is added to the graph, the adjacency arrays for vertices $u$ and $v$ are updated as before. To do this, each of their adjacency arrays must be resized to accommodate the new adjacent vertex id. After resizing the array, the index where the new id belongs is identified and all ids following the index are shifted to the next index, at which point, the new vertex id is inserted. Finally, the degree of the vertex is incremented by one to reflect the addition of the new edge. Each edge addition will require the dynamic adjacency array to be reallocated to a larger size array, which in the worst case, requires copying the entire array from the old memory location to the new. For that reason, similar to the previous data structures, the cost of edge updates in DAA is $O(\Delta)$.

### 4.2.4 Dynamic Intervalized Adjacency Array (DIAA)

DAA addresses many of the issues of the previous data structures. However, at best, that data structure will require no less memory than storing a static snapshot of the entire graph. As the size of graphs continues to rise, it is often desirable to use a structure whose memory requirements are less than $\Theta(|V| + |E|)$ in practice. It is for exactly this case that we explore two final data structures, the first of which we refer to as *Dynamic Intervalized Adjacency Array (DIAA)*. This data structure is a subtle variation of DAA, which provides a significant reduction in memory requirements for

certain classes of graphs.

The DIAA data structure is similar to DAA in that each vertex is stored as an object with a degree and an adjacency list representation. However, whereas in DAA, the adjacency list was represented explicitly using an adjacency array, in DIAA it is represented using an array of intervals and residual ids, see Appendix A. This simple optimization has the potential to reduce the memory requirements and improve the performance for a dynamic graph. If a graph exhibits locality, intervals will exist in vertex adjacency lists, and thus, memory requirements will be reduced.

The DIAA format can be specified as follows:

$$\delta\Big[i\big[E_1L_1\cdots E_iL_i\big]_{i>0}\big[R_1\cdots R_k\big]_{k>0}\Big]_{\delta>0}$$

where subscripted brackets denote the condition required for the enclosed portion of the format to be present, $i$ is the number of intervals, and $k = \delta - \sum_{j=1}^{i} L_j$, is the number of residuals.

Furthermore, by intervalizing the adjacency arrays, update and query operations can also be made more efficient. For example, when an edge is inserted into a DAA, the correct index for the new vertex id is identified and all elements subsequent to the index are shifted by one array index to make room for the new id. Contrast this with insertion into a DIAA. In a DIAA data structure the insertion of a new vertex id into a vertex object will have one of the following effects: 1) create a new interval, 2) extend an existing interval, or 3) become a residual. Examples of creating a new interval and extending an interval can be seen in Figures 4.3 and 4.4. There are three types of interval extensions, each of which is illustrated in Figure 4.4. The first type, *int-new*, happens when a new vertex id is added which is equal to one more or less then the highest or lowest id in an interval. In this case, the interval cannot be further extended by any of the residuals, which is exactly the second type, called *int-new-res*. The final type, called *int-int*, occurs when the new vertex id is exactly one less than the lowest id of one interval and exactly one more than the highest id of another interval.

Since an intervalized adjacency array can require at most $O(1)$ more space than its explicit counterpart, for storing the number of intervals in its vertex object, memory requirements for DIAA are expressed as $O(|V| + |E|)$. This also means that scanning and shifting of an intervalized adjacency array, which require at most the same number

Figure 4.3: Adding edge (13,15), which creates a new interval in each vertex object, shown in bold.

| Vertex | Degree | Intervals | Residuals |
|---|---|---|---|
| 13 | 9 | [1, 3], [8, 4], **[14, 2]** | |
| 15 | 6 | [32, 4], **[13, 2]** | |

Figure 4.4: Adding three edges, (4, 33), (4, 34), and (15, 31), to illustrate the three different interval extensions possible, changes denoted by bold text.

| Vertex | Degree | Intervals | Residuals | Type |
|---|---|---|---|---|
| 15 | 6 | **[31, 5]** | 14 | int-new |
| 33 | 7 | **[1, 5]** | 15, 23 | int-new-res |
| 34 | 7 | **[1, 6]** | 8 | int-int |

Figure 4.5: Compressed adjacency array, before integer coding, for example graph from Figure 4.1.

| Vertex | Degree | Intervals | Residuals |
|---|---|---|---|
| ... | ... | ... | ... |
| 13 | 8 | [1, 3], [8, 4] | 2 |
| 14 | 4 | | 1, 4, 4, 50 |
| 15 | 5 | [32, 4] | 1 |
| ... | ... | ... | ... |
| 32 | 8 | [2, 4], [10, 3] | 33 |
| 33 | 6 | [1, 3] | 55, 10, 8 |
| 34 | 6 | [1, 3], [5, 2] | 43 |
| ... | ... | ... | ... |

of memory operations as the explicit counterpart, incur a cost of $O(\Delta)$.

### 4.2.5 Dynamic Compressed Adjacency Array (DCAA)

It was shown in [36] that web graphs and even online social network graphs exhibit two particular properties which can be exploited to achieve compact representations. Namely, *similarity* and *locality*. Similarity states that given an appropriate ordering of the vertices of the graph, vertices which have ids that are close together in number, will have similar adjacency lists. Locality means that given the same ordering, the vertex ids in a given vertex' adjacency list should be close to the id of the vertex itself. Leveraging these two properties, the authors of [36] developed the WebGraph Framework for compression of web graphs.

In the case of web graphs, a lexicographic ordering of the vertices based on their corresponding URL produces an ordering that satisfies the similarity and locality properties. This is because most links contained in a web page are of navigational nature, i.e., they point to other pages within the same domain. This means that most links will exhibit locality, since the ordering of the pages lexicographically means that pages within the same domain will appear close in the ordering. Furthermore, since most navigational link structures are shared among several pages within a domain, the links associated with a group of pages sharing navigational structure will exhibit similarity. In practice, web graphs are typically ordered lexicographically for another more pragmatic reason, it allows for easy compression of the URLs, since URLs from the same domain appear close together when sorted lexicographically.

It is possible that the dynamic graph being represented does not exhibit the locality required to exploit an intervalized representation. It is also possible that the memory reduction achieved by intervalization alone is insufficient. In these cases, intervalization may not provide a satisfactory reduction in memory requirements, and an additional layer of compression applied on top of the intervalized representation may be attractive. To this end a final data structure is investigated. Referred to as *Dynamic Compressed Adjacency Array (DCAA)*, this data structure leverages ideas from the WebGraph framework described in Appendix A to achieve an even more compact representation than DIAA. While DCAA benefits from the locality required for a successful memory reduction in DIAA, it also has potential to reduce memory requirements for all classes of dynamic graphs, as it is based on differential encoding and a compact

representation of integer values.

In this representation, each vertex object stores the ids of adjacent vertices in a modified WebGraph compressed array. When a new edge $(u, v)$ is added to a DCAA, the initial behavior is much like that of the DIAA, i.e., two adjacency arrays are updated to reflect the addition of the new edge. However, in the case of DCAA, simply resizing and insertion is insufficient. Since the DCAA stores each adjacency array in compressed intervalized form, and the modified WebGraph compression does not allow for modification in compressed form, it is necessary to first decompress the intervalized adjacency array being updated. At this point, the new vertex id can be inserted following exactly the same procedure as DIAA. After insertion, the intervalized adjacency array is compressed again and the insertion is complete. The memory accesses required for the decompression, insertion, compression cycle is at worst, three scans over the entire adjacency array (case with no intervals), making the cost for edge update in DCAA to be $O(3\Delta)$.

### 4.2.6 Vertex Updates

The previous discussion addressed only the circumstances related to the dynamic update of the edges of a graph. In many cases, the range of vertex ids will not be known a priori. In these cases, the graph data structure must also support vertex updates. For LL, BCSR, and DAA, these operations are straight-forward to implement, since the same ideas which were used to support edge updates can easily be extended to vertex updates. The same is true of DIAA and DCAA, with one single, but important caveat. The caveat being that the memory and access efficiency of the DIAA and DCAA data structures depend highly on the ordering of the vertices. Under dynamic vertex updates, maintaining the optimal vertex ordering for these two data structures is a hard problem.

Since typical web and social graphs have significantly more edges than they do vertices, $|V| \ll |E|$, the number of edge updates will usually be at least an order of magnitude higher than the number of vertex updates. Therefore, it may be computationally feasible to periodically reorder the graph, assuming the existence of an efficient graph reordering technique designed to expose intervalization. The BFS reordering of [59] and the Grey code reordering of [37] both take linear time on the number of vertex and edges, and thus meet the requirements for the vertex update scheme. A slightly more

Table 4.2: Datasets used for experimental evaluation.

| Dataset | $\|V\|$ | $\|E\|$ | Type |
|---|---|---|---|
| in-2004 | 1382908 | 27182946 | web |
| eu-2005 | 862664 | 32276936 | web |
| lj-2008 | 5363260 | 39511571 | social |
| com-orkut | 3072441 | 234370166 | social |

computationally expensive approach, which has been shown to result in a more favorable vertex ordering for WebGraph based schemes is the Layered Label Propagation reordering of [38]. This method however, requires several iterations of a linear time reordering algorithm to achieve good results, thus should be reserved for the scenario when the maximum amount of compression is needed, regardless of the cost. Between reorderings, vertices can be assigned ids based on their chronological appearance. This introduces a trade-off between how often to reorder and the effectiveness of the intervalization mechanisms of DIAA and DCAA. Since vertex updates are infrequent compared with edge updates, a conservative approach with a smaller interval between reorderings can be employed to ensure intervalization effectiveness.

## 4.3 Experimental Design

**Datasets** We evaluated our algorithms using five real world datasets obtained from the Laboratory for Web Algorithmics, `http://law.di.unimi.it`. The first two datasets are examples of web graphs, while the latter two are online social graphs. Their characteristics are shown in Table 4.2. As can be seen, in all datasets, the number of edges is greater than the number of vertices by at least an order of magnitude.

**Implementation** All data structures and algorithms were implemented in C. The experiments were run with a single thread on an Intel (R) Xeon (C) 2.80GHz CPU. The test applications were compiled on Ubuntu 10.04 with GCC version 4.5.2. Timing was performed using the *gettimeofday()* POSIX.1-2001 function.

Through experimental evaluation, it was determined that the *malloc()* implementation shipped with the standard C library on Ubuntu 10.04 was unsuitable for the dynamic nature of many of the data structures described in this work. This is especially true for DAA, DIAA, and DCAA, whose memory cannot be bulk allocated like that which can be done when allocating nodes for a linked-list or the single allocation required to resize a static CSR structure. For this reason, we implemented our own malloc() library that is specifically designed for the way that memory is allocated for DAA, DIAA, and DCAA. Recall that the graphs which are being focused on here, namely web and online social graphs, tend to exhibit scale-free behavior. This means that the majority of vertices in the graph have a small degree. Since each adjacency list in DAA, DIAA, and DCAA requires a separate allocation, malloc() from the standard C library, incurs a great deal of bookkeeping overhead. This overhead was eliminated in the alternative malloc implementation by using the well known fixed-size block allocation technique instead of the linked-list management technique of many malloc implementations. By eliminating this memory overhead, the amount of virtual address consumed by an application utilizing dynamic data structures like those described here can be greatly reduced.

With regards to BCSR, we employ a policy for folding such that the LL gets incorporated into the static CSR only when a query algorithm is to be executed on the data structure. The advantage of this is that due to the good access characteristics of the static CSR, algorithms executed on the data structure after folding will be very efficient. The drawback of this approach is that the LL can grow quite large between algorithm executions. However, if algorithms are executed with sufficiently small intervals between, then the memory requirements of this approach will not grow undesirably high.

Finally, for the procedure of vertex reodering, we chose two strategies based on the class of graph under consideration. For web graphs, the lexicographic ordering of vertex URLs was used to impose an ordering on the vertices. This was shown in [33] to lead to a very compression friendly ordering, and is extremely efficient in implementation. For social graphs, an ordering based on the BFS ordering of [59] was used.

**Performance Assessment Metrics** For the experiments, each dataset was processed independently. Edges were chosen at random from each edge set $E$ to be added to the data structure being tested. Each edge was added exactly one time and edges were added in the same order across all data structures.

We measured the performance of the various approaches along three dimensions: (i) memory usage, (ii) update time, and (iii) access time. Memory usage was reported using two different metrics. The first was the amount of memory which each data structure occupied at any given time. This measure closely resembles the theoretical memory requirements, as it reports only the exact number of bytes needed to store each data structure. The second measure is virtual address space usage. This measure reflects the amount of memory obtained from the OS by the benchmark application to persist and operate on each data structure and is directly affected by the OS and malloc implementation's treatment of allocated memory.

The amount of time to update the graph data structures was measured as a function of the number of edges in the graph. The update execution time is reported as the aggregate number of milliseconds spent on edge addition since the last query operation.

**Benchmark Applications** To measure execution times for graph access, the following policy was used. The graph structures were queried after every $|V|/\alpha$ edge additions. For the results presented in Section 4.4, $\alpha$ was set to be 200. Each $|V|/\alpha$ interval will be furthermore referred to as a *snapshot*. Three kernel benchmarks were used to measure the graph access time. Each of the kernels was chosen to stress different graph access patterns. We believe that these three benchmarks are representative sample of graph kernels, which can be used as building blocks for more complex analysis.

The first kernel is the breadth-first traversal (BFT). BFT is an edge traversal which uses a queue to maintain the order in which to visit vertices. At each iteration, the edges of the current vertex are expanded and any vertices which have not already been enqueued, are added to the queue. The next iteration commences by popping the next vertex from the queue. If the queue is empty, but not all vertices have been visited, a random vertex is chosen which has not been visited. This is repeated until all vertices have been visited.

The second kernel is a neighbor query. The purpose of this query is to identify the set

of vertices which two vertices have in common and is implemented as a set intersection in practice. Typically, a set intersection requires iterating all the elements of each of the sets under consideration and testing for equality. If the sets are sorted, this can be done in a single pass over the two sets and only requires as many comparisons as there are elements is the smaller of the two sets. There are more complex algorithms for intersection of arrays, as opposed to linked-lists, which can be made extremely fast in practice, such as [60]. This operation can be further optimized for interval aware data structures like DIAA and DCAA, where instead of iterating over all adjacent edges of the two query vertices, processing proceeds as follows. Given two intervalized arrays, find the smallest element in each of the lists. This element can exist within the bounds of the first unprocessed interval or as the first unprocessed residual. Then, given the smallest element of each array, a comparison is made exactly as it would be for a standard set intersection. However, instead of iterating through each array comparing every element, when the elements of the two arrays are not equal, it is possible to skip entire intervals while only performing a comparison of their start id and end id, thus potentially saving a significant number of comparisons. Furthermore, two intervals can be intersected directly without iterating either of their ranges in its entirety.

The final kernel is the adjacency query. This is a simple check whether or not an adjacency list contains a specified vertex id. The standard approach is a linear search for linked-list and a binary search for arrays. However, like the neighbor query, adjacency queries can also be optimized for interval aware data structures. The idea is the same, instead of performing a binary search over every vertex id in an adjacency array, a modified binary search should be employed which makes a distinction between intervals and residuals and has the ability to treat intervals as singular elements.

In our experiments, the set of query vertices was kept consistent within each kernel benchmark. For example, in the BFT kernel, the same set of vertices was uses as the root of the BFT regardless of the data structure being tested. The same is true of the neighbor and adjacency queries.

Table 4.3: Number of intervalized edges in each graph dataset.

| Dataset | Intervalized Edges | % of Total Edges |
|---|---|---|
| in-2004 | 21386085 | 78.67 |
| eu-2005 | 22623251 | 70.09 |
| lj-2008 | 11288594 | 14.29 |
| com-orkut | 11083262 | 4.73 |

## 4.4 Results

### 4.4.1 Memory Requirements

Figures 4.6 and 4.7 show the memory requirements for each of the five dynamic graph data structures per dataset. From Figure 4.6, we see that the memory required by each data structure, follows the same pattern as the theoretical bounds for memory requirements for each data structure. As expected, intervalization and integer coding reduces the memory required in all cases. However, the scale of the reduction is much higher in the case of web graphs, where adjacency lists are typically well intervalized, as shown in Table 4.3.

Also interesting is the virtual memory required by each of the data structures, as can be seen in Figure 4.7. In the case of virtual memory, the footprint of each data structure follows the same pattern as requested memory, except for BCSR, which has the highest virtual memory requirements by nearly 20%. This can be attributed to the extra memory required to fold the linked-list portion of the graph into the static CSR, which is an important consideration if memory consumption during execution must be limited.

### 4.4.2 Update Time

The second measure used to evaluate the data structures was update time. Figure 4.8 shows the execution time for the addition of edges for each snapshot. Note that the execution time for a snapshot includes only those edges which were added during a

Table 4.4: Mean length (standard deviation) of linked-list being updated for each dataset under LL and BCSR data structures.

| Dataset | LL | | BCSR | |
|---|---|---|---|---|
| in-2004 | 555.76 | (1527.16) | 2.78 | (7.81) |
| eu-2005 | 1288.62 | (4806.19) | 6.44 | (24.16) |
| lj-2008 | 53.33 | ( 102.99) | 0.23 | (1.16) |
| com-orkut | 194.67 | ( 900.50) | 0.97 | (4.61) |

particular snapshot. As expected BCSR has the lowest edge addition time, when considered without the cost of folding. This is based on its reliance on linked-lists, which are very efficient for short lists. Since, BCSR offloads edges from its linked-lists into a static CSR occasionally, the linked-lists which it requires tend to be of shorter length, which can be seen in Table 4.4. However, when folding is considered, BCSR w/ fold, DIAA actually surpasses BCSR for sufficiently large and intervalized graphs, as would be expected based on the discussion in Section 4.2.4.

One might expect that DCAA would have the slowest edge addition time for all datasets, since the overhead of decompression and compression will have a negative impact on addition time. However, as Figure 4.8 shows, DCAA is actually faster than LL at later snapshots. As mentioned in Section 4.2.4 and supported by Figure 4.8 (DAA vs. DIAA), adding an edge into an intervalized adjacency array is at least as efficient as inserting into an explicit adjacency array. Since ordered insertion into a linked-list is slower than insertion into an array (LL vs. DAA) and the web graphs are highly intervalized, the efficiency introduced by intervalization allows the DCAA data structure to support edge additions at a higher rate than LL. The same is true for the social graphs, but to a lesser degree. However, even for intervalized datasets, DCAA still requires more time for edge addition than the rest of the data structures.

### 4.4.3 Performance of Benchmark Applications

The results for the three kernel benchmarks discussed in Section 4.3 are shown in Figures 4.9 to 4.11, respectively.

Figure 4.9 shows the results of the BFT kernel benchmark, where it can be seen that BCSR exhibits the fastest overall execution time, with DAA and DIAA close behind. This is exactly as we would expect, since the representation of the graph during operation in the BCSR data structure is in static CSR format. Even with folding time considered, BCSR is still fastest, which is a consequence of the nature of the query, namely, a traversal of the entire graph. Since graph traversal in BCSR has the same complexity as edge folding, but a less friendly memory access pattern, we would expect that the graph traversal cost will outweigh the cost of edge folding.

The DAA data structure stores the graph in conceptually the same way as BCSR, but because each of the adjacency lists in DAA is allocated dynamically, the data locality that can be exploited in BCSR is not present to the same degree in DAA. The same is true of DIAA, with the addition of a slight overhead due to the bookkeeping of the intervalized adjacency lists and DCAA with a further overhead of integer coding. DCAA executes between 3.5 and 4.5 times slower than BCSR, which can be attributed to the overhead of decompression of the adjacency lists before operation. Finally, LL consistently reports the slowest execution times for BFT.

In the case of the neighbors query, a set intersection is performed between two randomly chosen vertices. As Figure 4.10 shows, the results of performing a set intersection are similar for LL, BCSR, and DAA, but quite different when compared with the results for BFT for the other two data structures. As alluded to above, this is due to optimizations which can be exploited by the interval aware data structures. In any of the first three data structures, a basic set intersection requires iterating all the edges of at least one of the participating vertices. In the case that the vertex degrees of the two vertices being intersected are different, only the adjacency list of the smaller degree vertex need be fully iterated. However, in the final two data structures, namely DIAA and DCAA, adjacency lists are stored as intervals and residuals. In these cases, rather than iterating entire adjacency lists, it is only necessary to iterate the intervals and residuals. If the adjacency lists are well intervalized, then the number of intervals and residuals will be much less than the degree of the vertex, meaning a set intersection can be performed with many fewer comparisons.

For web graphs, which are well intervalized, this results in in DIAA having the fastest execution time for neighbor queries. Further, the access time of DCAA also comes very

close to surpassing that of the non-intervalized data structures. This illustrates an important characteristic of the two classes of data structures. Namely, that those data structures which are not interval aware will have access times which are non-decreasing as a function of the number of edges in the graph, while interval aware data structures will exhibit decreasing access times as the number of edges grows and the intervals become more dense, assuming an interval friendly ordering of the vertices.

The final query type tested was the adjacency query. At first glance, an adjacency query is similar to a neighbors query, since both require the scanning of particular adjacency lists. However, just as neighbors queries can be optimized to leverage an intervalized data structure, so too can adjacency queries. This is especially true in the case of the DCAA data structure. Instead of decompressing the entire array then performing a binary search, which costs $O(n + \log n)$, each element of the array can be checked as it is decompressed and the search can return as soon as the desired element is found, without decompressing the rest of the array. The cost of the optimized search is $O(n)$, which is slightly better than decompressing and performing a binary search. As Figure 4.11 shows, this type of optimizations means that the DCAA data structure is very competitive in terms of adjacency query execution time. It is approximately 3x slower at worst, and less than 2x slower in most cases.

### 4.4.4 Selection of $\alpha$ for BCSR

There is an inherent trade-off to be made if an $\alpha$ based policy is chosen for edge folding in BCSR. Fold too often and the cost of the edge folding operation outweighs the benefits it gives in terms of graph update and access times. Figure 4.12 shows that there is a sweet-spot in terms of $\alpha$. For the web graph dataset, eu-2005, the best $\alpha$ value in terms of total benchmark suite execution time is 50. For the social graph dataset, com-orkut, this value is 10. The most likely explanation for the effectiveness of small $\alpha$ for social graphs is due to their relatively small linked-list lengths, as shown in Table 4.4. Since the linked-lists are short, relative to those of the web graphs, the advantages of frequent folding are reduced, since one of the main goals of edge folding is to reduce the lengths of the linked lists.

### 4.4.5 Results on All Graphs

In our experiments, eu-2005 and com-orkut were representative of web graphs and social graphs respectively. For this reason and due to space limitations, detailed plots for the remaining three datasets have been omitted. However, Figure 4.13 shows a summary of benchmark suite execution time for each dataset. This figure supports conclusions drawn from the previously presented results in all cases except one, namely that BCSR is inferior to DIAA in terms of access time and query time. Figure 4.13 shows that the total execution time for the benchmark suite is smaller for BCSR than DAA. This is due to the fact that Figure 4.13 shows the total execution time for the benchmark suite and does not illustrate the following fact in the context of web graphs. In early snapshots, DIAA is slower, due to bookkeeping overhead, but in later snapshots, the intervalized structure of the web graphs, allows the update and access time to decrease in DIAA versus, a non-decreasing update and access time for BCSR, recall Figures 4.8 to 4.11.

Figure 4.6: Required memory for each data structure.

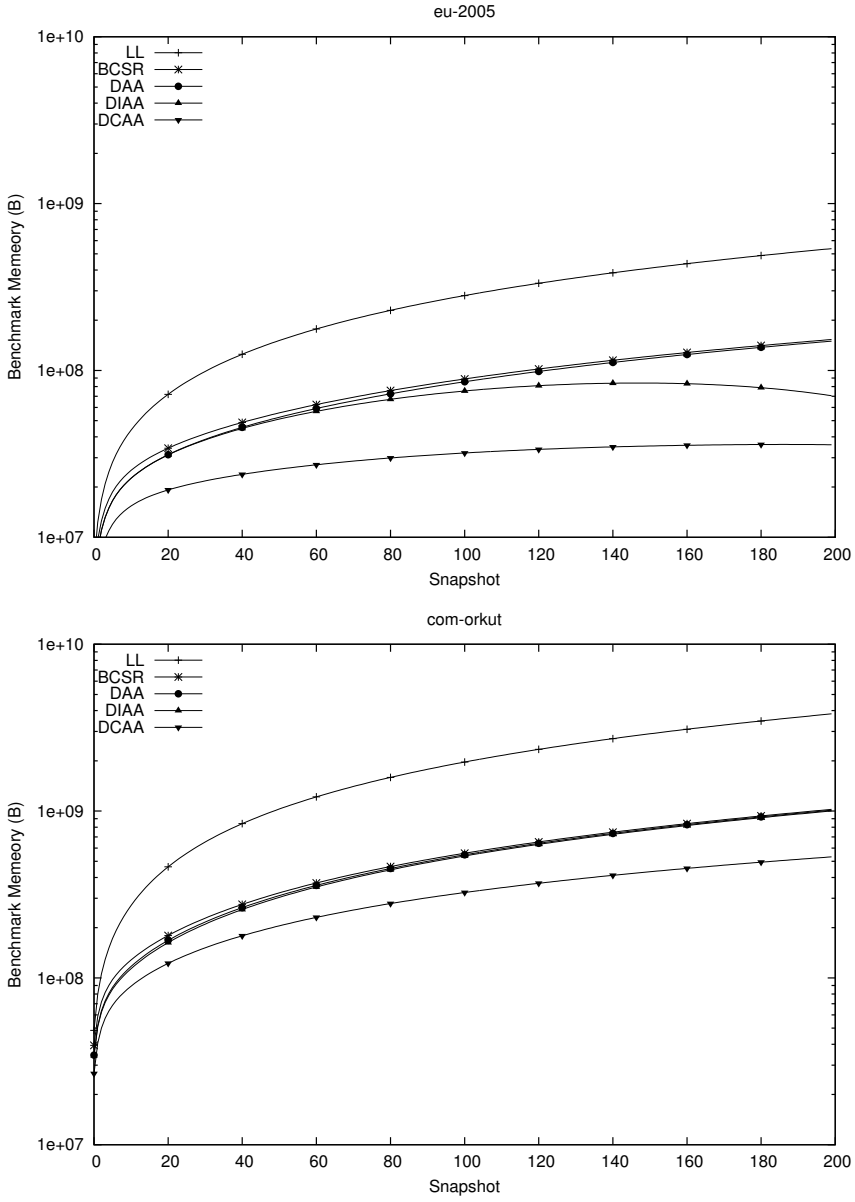


Figure 4.7: Virtual memory usage during benchmark application.

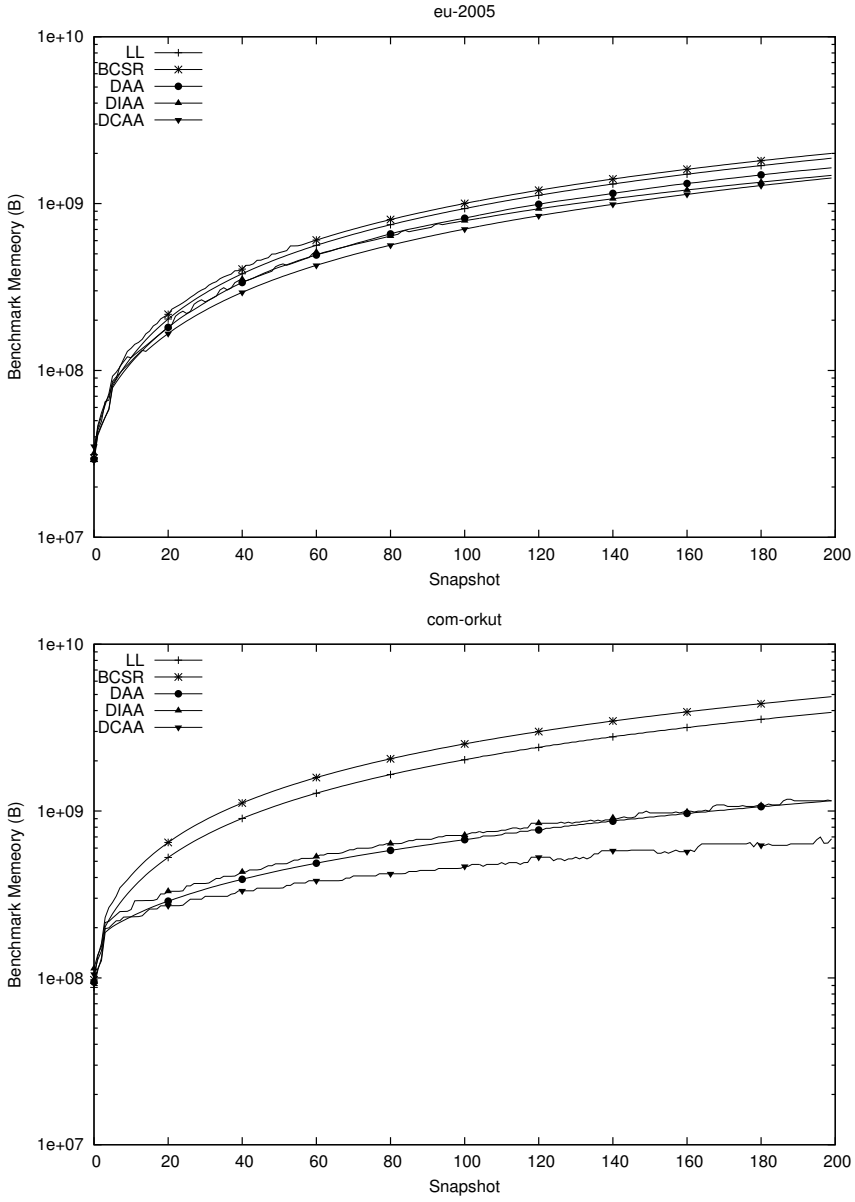


Figure 4.8: Edge addition times.

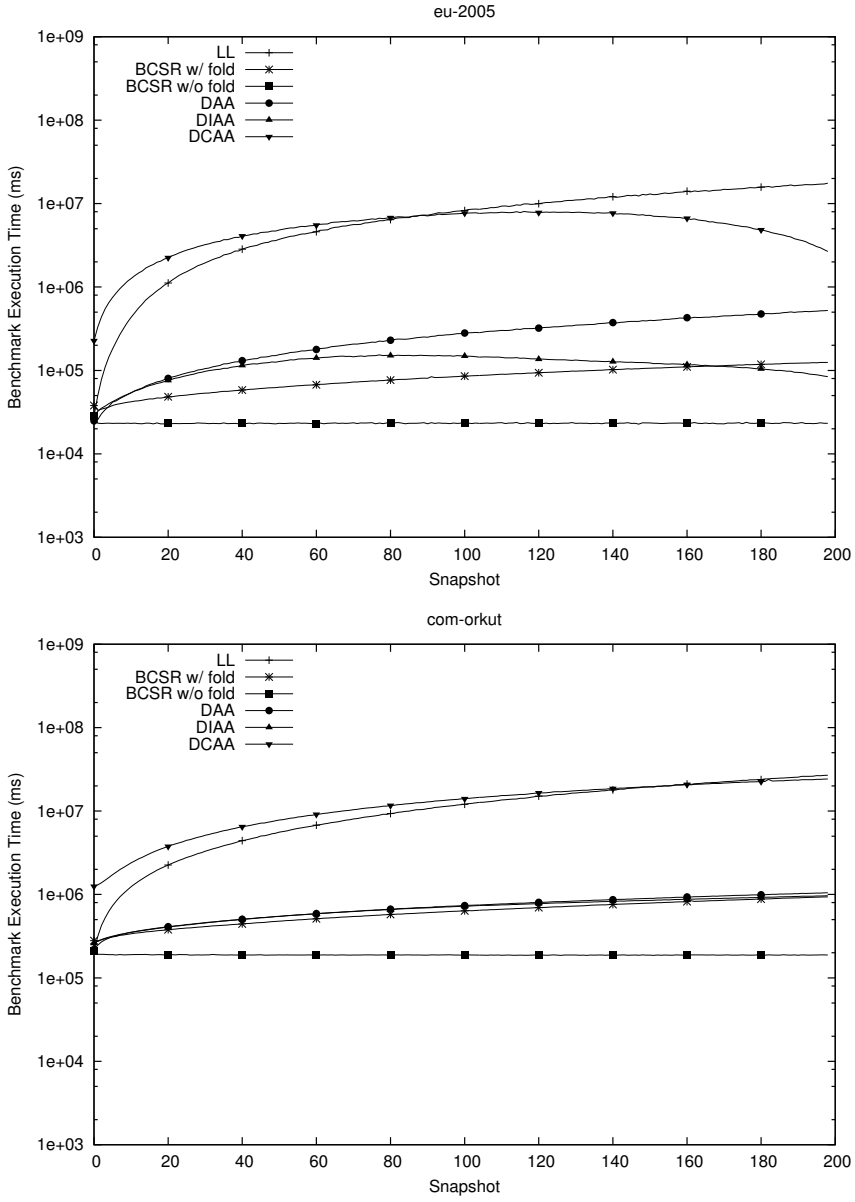


Figure 4.9: Breadth-first traversal results.

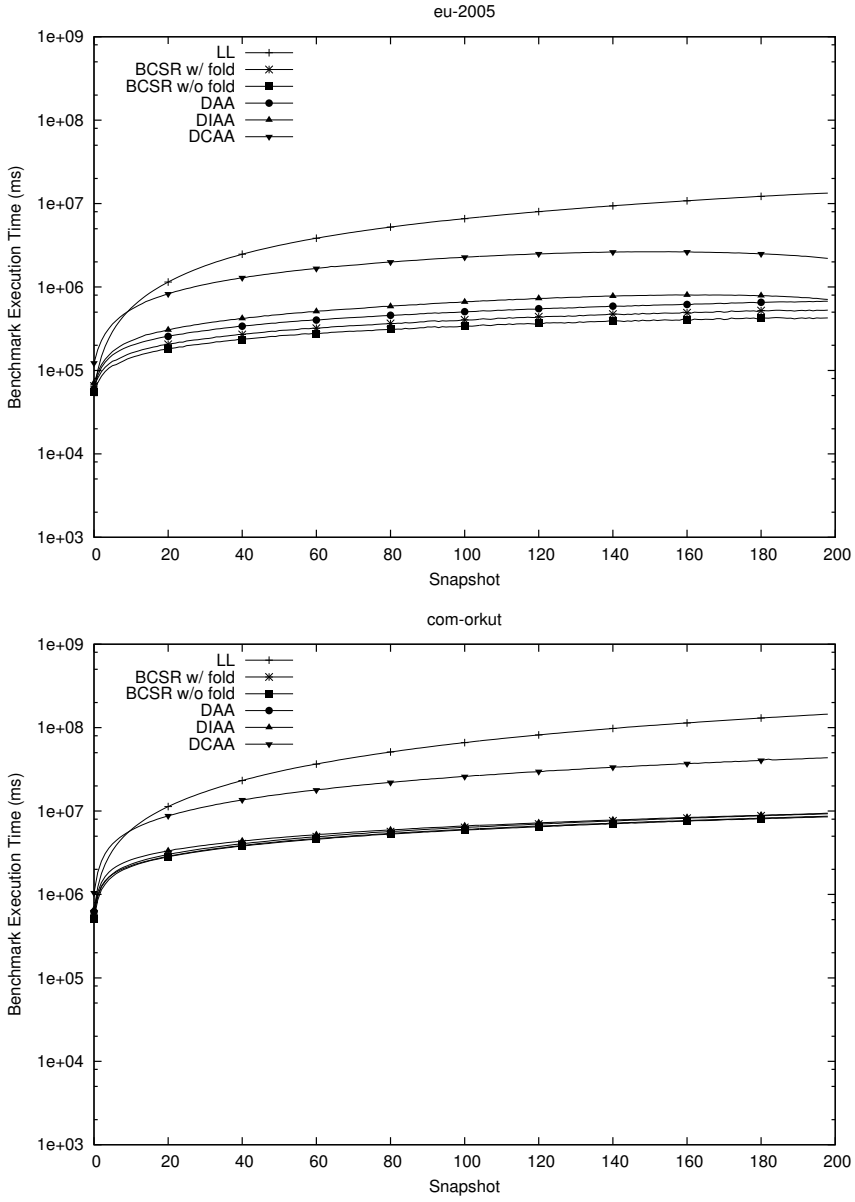


Figure 4.10: Neighbor query results.

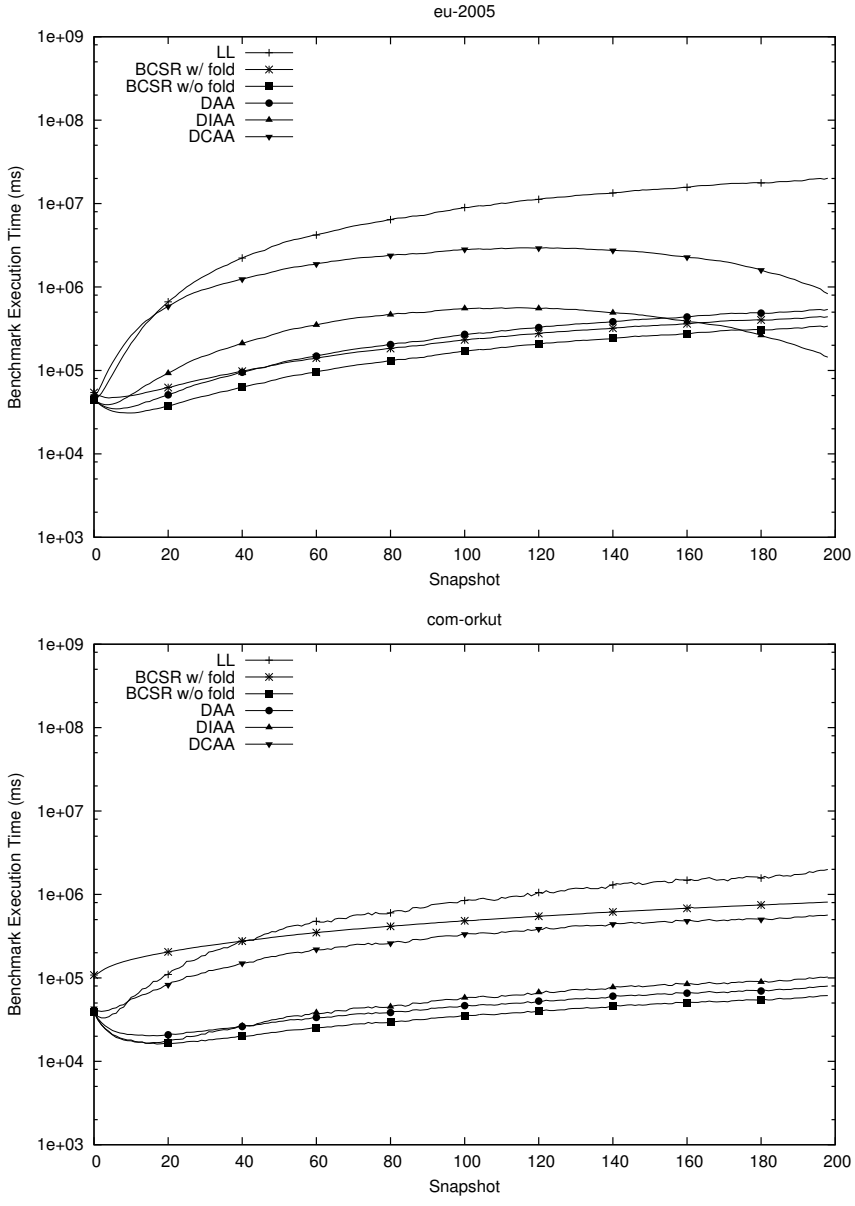


Figure 4.11: Adjacency query results.

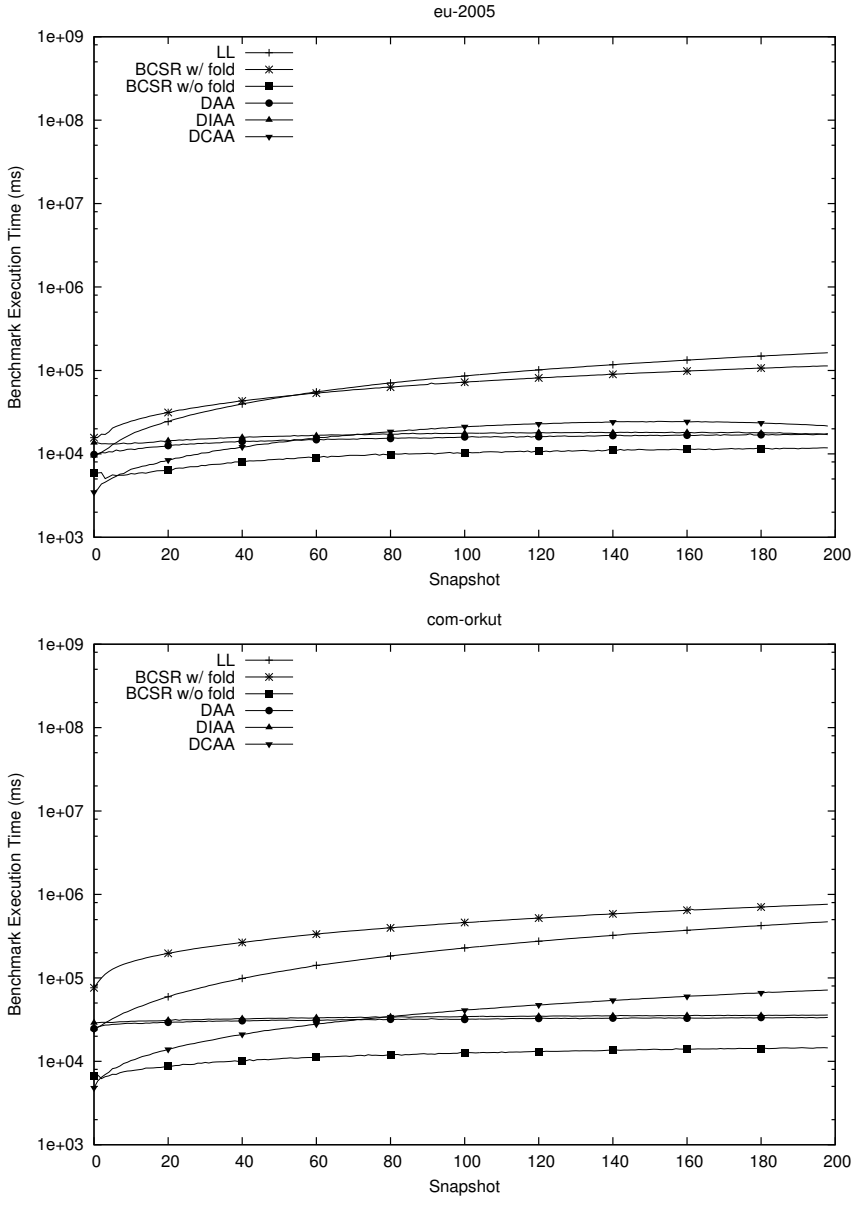


Figure 4.12: Benchmark suite execution time under variations of $\alpha$.

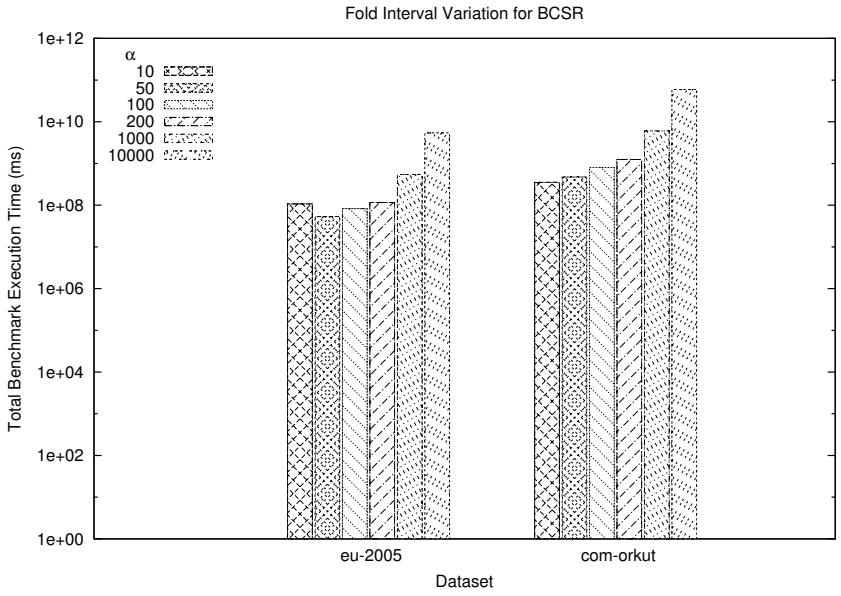
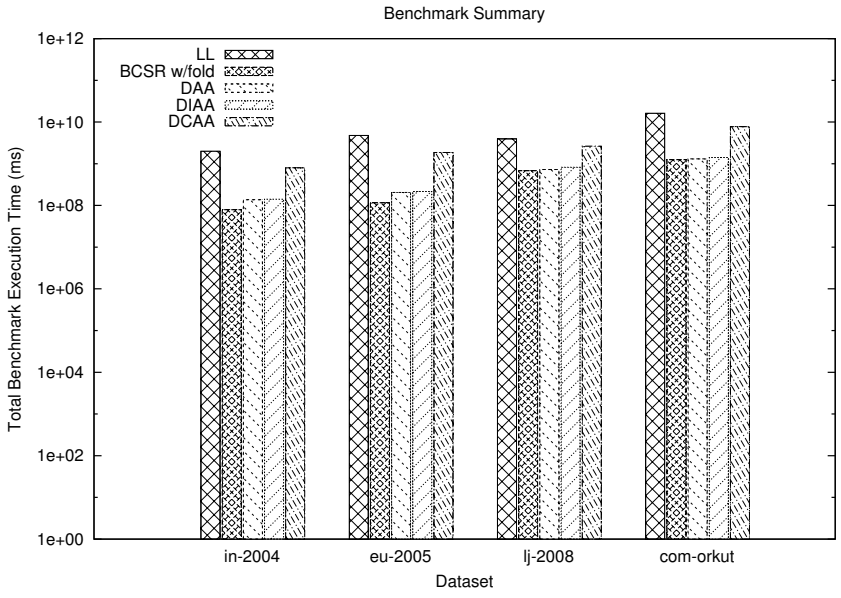



Figure 4.13: Execution time required for benchmark suite for each data structure.



# Part II

# External Memory-Based Methods

# Chapter 5

# Preliminaries

## 5.1 Background

### 5.1.1 Virtual Memory

Most modern *operating systems (OSs)* use *virtual memory (VM)* to give the illusion to developers that a given process has a contiguous memory address space not shared by any other process, and that the size of said address space is not necessarily limited by the amount of physical memory (DRAM) present in the system. In this context, process simply refers to the independent entity that an OS assigns to instance of an application being executed. To accomplish this, the OS provides each process in the system with a private and unique address space, known as a *virtual address space*. The locations of each process' virtual address space in use at a particular point in time are mapped to the *physical address space*, which is the address space that belongs to the physical memory and is shared by all processes in the system. It is the responsibility of the OS, in coordination with some hardware capabilities, to maintain the correct translations of virtual memory addresses to their corresponding physical addresses. Since the virtual address space of each process is not limited to the available physical memory, it is also the responsibility of the OS to manage the over-subscription of physical memory.

This functionality is achieved using a technique known as *paging*. In an OS that supports paging, memory addresses are accessed at a granularity of a *page*, hence the name paging. Each time a process requests access to a virtual memory address, the

virtual address is translated to a physical address. If one such address exists, then the page that it belongs to is considered *resident* and the access proceeds. When the physical memory of the system has been over-subscribed, it is possible that no such physical address exists. This occurs when the physical memory that was mapped to a virtual address has been re-allocated to another virtual address. In this case, the former virtual address has been *evicted*. To preserve the data contained in a page of memory when its virtual address gets evicted, the OS stores it to some secondary storage medium, known as the *swap space*, before eviction. If an access is requested for a virtual address that has been evicted, then a *page-fault* occurs and the OS must map the requested virtual address to some physical address and populate the physical memory with the data previously stored to the swap space. If all physical memory has been mapped to a virtual address, then the OS must choose a resident virtual address and evict it. The choice of which resident virtual address to evict is known as a *page replacement policy*.

While paging allows for the processes running on a system to over-subscribe the physical memory, it comes at a cost. Each time that a page-fault occurs, the OS must read from and write to whatever storage medium has been designated as the swap space. Since the swap space allows for over-subscription of physical memory (DRAM), it is traditionally a device which has a much higher capacity than DRAM, often a hard disk drive or a solid-state drive. The consequence of this secondary storage access is a dramatic increase in memory access times, since these devices are known to have much higher access times than DRAM.

### 5.1.2 Multitasking

Arguably, one of the most important evolutions of computer operating systems was the notion of multitasking. A successor to early multiprogramming systems, multitasking OSs are able to coordinate the execution of several programs (tasks) "at the same time". Often, the programs, which are said to be executing *concurrently*, must share the hardware resources available in the system, most notably CPU cycles. Sharing of such resources has traditionally been achieved in one of two ways.

In the first and older of the two approaches, known as *cooperative multitasking*, the programs running concurrently, voluntarily yield access to hardware resources. The

voluntary nature of this approach makes it unsuitable for general purpose OSs, since a single program, who does not with to yield resources, can halt progress for all other programs running on the system.

The modern alternative to cooperative multitasking is *preemptive multitasking*, where programs are interrupted according to some scheduling policy and required to yield system resources so that other programs may receive access. This allows for much more predictable system behavior, since every programming running concurrently is guaranteed some amount of access to system resources. For this reason, it is much better suited for general purpose operating systems and in fact is the standard for most mainstream OSs like Windows, Linux, and MacOS.

If the programs that are executing do not require access to shared resources, as may be the case in a multi-core CPU or a distributed system, then the programs are said to be executing in *parallel*. There is an important semantic distinction to be made between concurrency and parallelism. Namely, concurrency does not imply that the programs execution is simultaneous, as in the same instant in time, only that their execution is overlapped, while parallelism does. This has implications for later discussions because the Linux kernel, as do most contemporary OSs, uses preemptive multitasking to control access to shared resources.

### 5.1.3 MPI: Message-Passing Interface

The Message-Passing Interface (MPI) standard is a standard for a parallel programming model known as message-passing [61]. The message-passing model targets machines that consist of processing elements that have distinct address spaces, and thus must rely on sending message to communicate with each other. This closely relates to distributed-memory architectures, where such a programming model is a natural choice, but it is not limited to only such systems.

Th MPI standard defines two simple operations upon which nearly every other operation relies, namely send and receive. If one processing element would like to access data stored in the address space of another processing element, the processing elements must send / receive messages to / from each other to copy the desired data from one address space to the other. Because the task of explicitly moving data can be tedious and experience has taught us they there are some communication patterns

that reappear in many parallel algorithms, the MPI standard also defines a large set of common communications operations, some of which are included in Table B.1.

Programs written using the functionalities specified in the MPI standard are often of asynchronous nature. By this we mean that during execution, the processing elements only synchronize when some sort of interaction must take place. This could be to exchange data or as to indicate that a processing element has reached a certain point in its execution. Between interactions, processing elements compute completely independently of each other, without the need for any communication. The reason for this is that communication itself has no computational value, i.e., it alone does not contribute to the solution of the problem being solved. As such, MPI programs are typically engineered to minimize communication while maximizing the amount of independent computation. This is a common characteristic of a another parallel programming model known as the *Bulk Synchronous Parallel* model [62], which explicitly takes into consideration the cost of communication and synchronization. There are many high quality implementations of the MPI standard available both freely and commercially [63–66].

### 5.1.4 OpenMP: Open Multi-Processing

Open Multi-Processing (OpenMP) is a parallel programming model that targets shared memory multi-processing computer architectures [67]. The most significant contribution of OpenMP over other shared memory programming models is that it is cross-platform by design. OpenMP is not a particular implementation or library, rather, it is an *application programming interface* (API) specification that defines a simple programming model and an accompanying interface that programmers can use to develop parallel programs. In practice, OpenMP manifests as a set of compiler directives or pragmas. In the C language, this takes the following form

```
#pragma omp directive [clause],
```

which indicates to the compiler that the code block following the directive should be compiling according to the language extensions defined in the OpenMP specification. Some possible directives are listed in Table C.1 and some possible clauses in Table C.2. An example OpenMP function to compute the dot-product of two vectors is shown

in Listing D.1.

The basic execution model of OpenMP is the fork/join model. In this model, the program begins with a single thread of execution, called the *master*. A thread of execution is commonly referred to simply as a *thread*, although this does not imply a dependence or relationship to the OS concept of a thread. When the master thread reaches a parallel region, identified by the `PARALLEL` directive, new threads of execution are *forked* and computation progresses until reaching the end of the code block enclosed by the directive. At this point, the independent threads are *joined* again into a single thread of execution. Again, the choice of terminology used by OpenMP, namely fork and join, imply no dependence on the OS functionalities `fork()` and `join()` present in most modern OSs. Within a parallel directive, it is often desirable for the threads to perform work independently of each other. This is easily expressed in OpenMP using any of the worksharing clauses, like `FOR` or `SECTION`. The `FOR` directive distributes the iterations of a `for`-loop to the threads according to the argument specified to the `SCHEDULE` clause. The `SECTION` directives identifies a block of code that can be executed in parallel with other blocks designated with the `SECTION` directive within the same `SECTIONS` block. It is possible to combine the `PARALLEL` and `FOR` directives into a single `PARALLEL FOR` directive in the case that the parallel region contains a single `for`-loop. Finally, within a parallel region, the program developer might want the threads to synchronize in some way, but not cease to exist. In this case, directives such as `BARRIER` or `ATOMIC` are useful.

The basic memory model of OpenMP is the shared memory model. This means that when there are multiple threads of execution, they all have the same view of the memory space. As a consequence, the threads can manipulate the data stored and accessed by other threads. This provides a very efficient means by which independent threads can communicate and is one of the advantages of the shared memory model.

## 5.2 Literature Reviews

### 5.2.1 Out-of-Core Computation

There are a variety of research efforts related to efficient memory management strategies for data intensive out-of-core applications. Most of the previous work can be broadly

categorized as follows: (1) OS memory management policies, (2) problem specific out-of-core solutions, or (3) efficient general purpose data intensive solutions.

Many page replacement policy studies have been performed under various conditions and for various workloads. Some examples are Lee et al. [10], Park et al. [14], and Qureshi et al. [15]. In each of their respective papers, the authors demonstrated different circumstances that application performance can be improved using page replacement policies designed for the characteristics of the application. Other works, like that of Engler et al. [7], focus on the design and implementation of a general purpose virtual memory subsystem at the application level. The authors argue the efficacy of an application level VMM and included micro-benchmarks to show its feasibility. Our approach introduced in Chapter 6 falls into this broad category of application level VMM.

More recently, there has been an emergence of interest in memory management systems catered toward out-of-core applications. A large amount of this work has been focused on problem specific out-of-core solutions. Typically, this means that a memory management strategy is tailored for the specific characteristics of the problem. While this level of customization generally yields the best results, the engineering cost is prohibitively high for most applications. Thus, this type of research has been focused mainly on those problems where performance is critical. For example linear algebra kernels [16] and data management systems [6].

There has also been a number of projects related to optimized general purpose virtual memory management systems. Most relevant to our work are three projects. The first is by Brown et al. [5], who looked at an entirely compiler-based solution for prefetching and releasing memory in out-of-core applications. The compiler used analysis of the source code to identify potentials points in the execution where data could be safely loaded and unloaded so as to reduce memory access latency. The complete solution also included a runtime layer which intercepted the compilers prefetch and release instructions and decided if they were still appropriate to execute based on the current state of the system at the time of interception. Using this approach I/O stall time could be reduced by more than 50% for many large scientific applications [5]. The second is the DI-MMAP system of Van Essen et al. [8,17], a kernel module which replaces the existing Linux memory map runtime with a custom runtime. For the metagenomics application used in the paper, the DI-MMAP system achieved a 4.88× performance improvement over the Linux kernel

memory map runtime. Like SBMA, DI-MMAP is also built as a drop in replacement for memory acquisition, however DI-MMAP is built as a kernel module and is seen as an extension to the kernel rather than an application level VMM. In contrast to the Linux kernel VMM, which is optimized for shared libraries, DI-MMAP is optimized specifically for data-intensive out-of-core applications, including optimizations like bulk flushing of the page table and transition-look-aside buffers. Compared with SBMA and OpenOOC, the DI-MMAP system is much lower-level and thus, has no notion of the cooperative multi-tasking nature of BDMPI or OpenMP. Lastly is the work of Meswani et al. [13], which addresses this problem for future Exascale machines. The authors argue that a programmer-driven approach to memory management is the quickest way to reach the bandwidth targets of the Exascale machines. Their results show that while programmer-driven methods are an improvement over automated systems, their is still more research to be done before the bandwidth targets of the Exascale machines are reached.

Orthogonal to the previously mentioned works, but still within the category of general purpose solutions for out-of-core computations is a collection of projects focused on the development of frameworks for expressing computations in a way that can be easily executed out-of-core in distributed systems. Some of the most significant outcomes of this work have been the functional programming model used by the MapReduce [68] and associated frameworks (e.g. Hadoop [9] and Spark [12]) and the vertex centric model used by various graph-based distributed processing frameworks (e.g. Pregel [11], Hama [69], GraphLab [70], and Giraph [71]).

# Chapter 6

# Distributed Memory Architecture

## 6.1 Methods

### 6.1.1 BDMPI: BigData MPI

*BigData MPI (BDMPI)* is a solution to efficient and transparent out-of-core execution for distributed memory systems, introduced by Lasalle et al. [72]. The key idea behind BDMPI is that the programmer needs only to develop a memory-scalable parallel MPI program by assuming that the underlying computational system has enough computational nodes to allow for the in-memory execution of the computations. In a memory constrained system, this program is then over-decomposed into a sufficiently large number of processes so that the per-process memory fits within the physical memory available on the underlying computational node(s). BDMPI maps one or more of these processes to the computational nodes and relies on the OS's VMM to accommodate the aggregate amount of memory required by them. BDMPI prevents memory thrashing by coordinating the execution of these processes using node-level cooperative multi-tasking, which limits the number of processes that can be running at any given time. This ensures that the currently running process(es) can establish and retain memory residency and thus achieve efficient execution. BDMPI exploits the natural blocking points that exist in MPI programs to transparently schedule the cooperative execution of the different processes.

BDMPI is implemented as a layer between an MPI program and any of the existing

implementations of MPI. From the application's perspective, BDMPI is just another implementation of a subset of the MPI 3 specification. Programmers familiar with MPI can use it right away and any programs using the subset of MPI functions that have been currently implemented in BDMPI can be linked against it unmodified.

**Master and Slave Processes** The execution of a BDMPI program creates two sets of processes. The first is the MPI processes associated with the program being executed, which within BDMPI, are referred to as the *slave* processes. The second is a set of processes, one on each compute node, that are referred to as the *master* processes. The master processes are at the heart of BDMPI's execution as they spawn the slaves, coordinate their execution, service communication requests, perform synchronization, and manage communicators.

**Node-level Cooperative Multi-Tasking** The master maintains information about the state of the various slaves, which they can be in one of the following states: currently running, blocked due to an MPI receive operation, blocked due to an MPI collective operation), scheduled for execution if resources when resources become available, and finalized.

The blocking/resumption of the slaves is done jointly by the implementation of the MPI functions in the slave library and the master processes. If a slave calls an MPI function that cannot be satisfied immediately, it notifies its master and then blocks by waiting for a message from the master process. The master updates the state of the slave to blocked and proceeds to select another runnable slave to resume its execution by sending a message to it. When a slave receives a message from the master, it proceeds to complete the MPI function that resulted in its blocking and returns execution to the user's program. If more than one slave is at a runnable state, the master selects for resumption the slave that has the highest fraction of its virtual memory already mapped on the physical memory. This is done to minimize the cost of establishing memory residency of the resumed slave.

Since all communication/synchronization paths between the slaves go via their masters, each master knows when the conditions that led to the blocking of one of its slaves may have changed and modify their state from blocked to runnable.

### 6.1.2 SBMA: Storage Backed Memory Allocation

Even though BDMPI's execution model is designed to maximize the amount of work that can be done with the data that was fetched from disk, the experiments in [72] showed that further performance improvements can be obtained by explicitly controlling how resident memory is transferred to and from disk. These optimizations require an in-depth understanding of the execution and memory models of the BDMPI runtime. Thus, it would be desirable to have a runtime system that can leverage the cooperative multi-tasking execution model of BDMPI to perform many of these optimizations automatically.

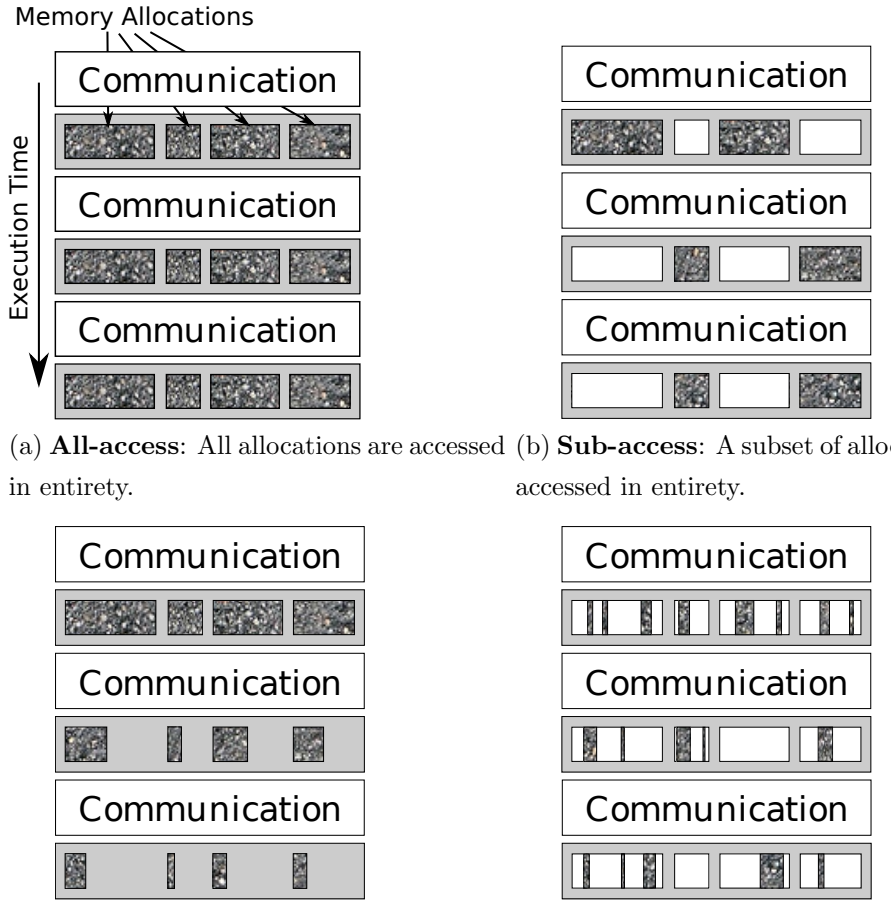
Motivated by the observation that optimized resident memory exchange can be implemented without any burden to application developers, we created a new VMM subsystem for the BDMPI runtime which incorporates knowledge of the system's execution and memory models. We call the new VMM subsystem *Storage-Backed Memory Allocation* (SBMA), for its use of file backed memory mappings to persist data between exchanges of resident memory.

SBMA is implemented as an application layer VMM and is relied on by the BDMPI runtime in place of the OS VMM. Therefore, it is the responsibility of SBMA to coordinate the transfer of data between physical memory and disk whenever necessary. Furthermore, since SBMA is intended to be used as a drop-in replacement for the OS VMM, this must be done in such a way that the application's memory mappings remain unchanged regardless of the mappings' residency in physical memory. By being cognizant of the BDMPI execution and memory models, SBMA is able to facilitate the exchange of resident memory in a manner that reduces interaction with the disk beyond the capabilities of the OS VMM.

**Targeted Memory Access Patterns** To guide the design decisions of the SBMA architecture, we identified four memory access patterns that characterize a wide range of programs and simultaneously satisfy the assumptions of the BDMPI execution model. Figure 6.1 illustrates each of these memory access patterns. The figures assume that the size of physical memory is roughly equal to the aggregate amount of memory allocations from the first execution phase shown.

The first (Figure 6.1a) of these access patterns, hereafter referred to as *all-access*,

Figure 6.1: Four common memory access patterns. Each sub-figure shows a selection of a single slave process' execution phases (communication followed by computation). Within a computation phase, memory allocations are represented by separate blocks and accesses to an allocation are indicated by a filled block or section.




(a) **All-access**: All allocations are accessed in entirety.

(b) **Sub-access**: A subset of allocations are accessed in entirety.

(c) **Mod-access**: Some allocations change size in successive phases.

(d) **Part-access**: Allocations are partly accessed.

results from an application that accesses all of its memory allocations in their entirety during each of its execution phases. An access pattern like this can benefit from bulk loading and unloading of its entire virtual memory space, since it will all be required for each execution phase. A similar access pattern, shown in Figure 6.1b and referred to as *sub-access*, is an application that only accesses a subset of its allocations, but still

accesses them in their entirety during a given execution phase. In this case, aggressively loading/unloading all of the virtual memory associated with a process, may lead to excessive disk interaction. Rather, it would be preferable to bulk load/unload entire allocations instead of a process' entire memory space. The first two access patterns are static in terms of the size of allocations across execution phases and only change in how these allocations are accessed. In many cases, the memory footprint of a process is not fixed throughout the entire application. The memory access pattern, known as *mod-access* and shown in Figure 6.1c is representative of such an application. Here, the size of the allocations being accessed during successive execution phases changes. In the figure, the allocations are shrinking, but they could also be growing instead. For such a memory access pattern, it is not always necessary to load/unload resident memory between every execution phases. During execution phases that access a set of sufficiently small allocations, it may be possible for allocations to remain resident in physical memory between blocking points. The final (Figure 6.1d) and most general, called *part-access* of the memory access patterns is the case where allocations are not necessarily accessed in their entirety. Consider as an example, an application which has a brief exchange of point-to-point communications which update only a small part of some number of allocations. For such an access pattern, it is not desirable to bulk load/unload allocations, since this will almost certainly lead to unnecessary disk interaction for one case or both. Instead chunks of the allocations should be loaded/unloaded on demand to reduce the amount of data read to/written from disk.

### 6.1.3 SBMA Architecture

To provide these functionalities, SBMA operates in the following way. Each time that a slave process allocates memory, the request is handled by SBMA, which obtains the memory from the OS on behalf of the slave process. Internally, SBMA keeps track of which parts of the allocation are currently resident in physical memory and which have been modified by the application. This way, just before the physical memory becomes over-subscribed, SBMA exploits the cooperative multi-tasking nature of the BDMPI runtime and evicts the memory of blocked slave processes at an allocation granularity. Like a traditional VMM, SBMA uses a file on disk to persist the contents of evicted memory. In addition, by tracking the parts of each allocation that have been modified

by the application, only those parts which have been modified since last being written to disk need to be written upon eviction. Likewise, when an allocation is re-admitted into physical memory, only those parts which exist in the file on disk need to be read. To ensure that memory mappings remain valid from acquisition to release, despite being evicted and re-admitted into physical memory, SBMA relies on a core set of memory-related functionalities provided by the Linux OS, discussed in detail in Section 6.1.4.

To reduce unnecessary data transfer between physical memory and disk, SBMA exploits the semantics of certain MPI functions. When a slave process blocks on a MPI operation that requires it to transfer data (e.g., point-to-point communication or collective operations), it is within the semantics of such an operation to *discard* the contents of any output buffers before receiving data. This means that the BDMPI runtime may forgo writing to file any memory regions marked as modified which are present in the output buffers of such MPI routines.

Within this framework, we developed three different strategies for managing the exchange of resident memory, the details of which are provided in the rest of this section.

**Aggressive Read / Aggressive Write (ARAW)** The first strategy is directly related to the explicit file I/O optimization described in [72]. This strategy is designed for applications with an all-access or part-access memory access pattern like those shown in Figures 6.1a and 6.1b. In this approach, when a process enters a blocking state, it writes any modified parts of its allocations to the appropriate file on disk and then releases the associated physical memory resources, a procedure which we call *unloading*. Upon exit of the blocking state, the first access to each allocation causes physical memory to be acquired for it and any parts of the allocation which exist in its associated file to be read, a procedure that we refer to as *loading*. Note that only those allocations that a slave process accesses during an execution phase are actually loaded, see Figure 6.1b. Furthermore, ARAW requires no communication between the master process and the slave processes in order to coordinate the exchange of resident memory.

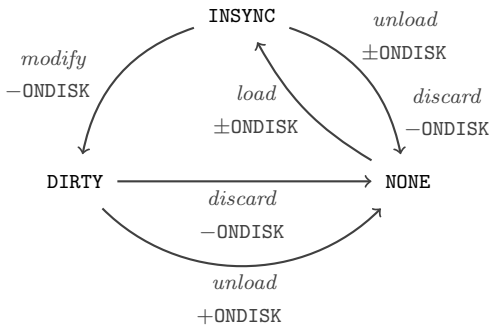
**Aggressive Read / Lazy Write (ARLW)** The are many applications that access their allocations in their entirety, but do not access a sufficient number of their allocations to over-subscribe physical memory during every execution phase. Such applications may have sub-access (Figure 6.1b) or mod-access (Figure 6.1c) memory access patterns.

Motivated by this, the Aggressive Read / Lazy Write (ARLW) approach allows memory to remain resident until memory becomes over-subscribed. By unloading allocations only when required, interaction with disk during these types of execution phases can be significantly reduced and in some cases, completely avoided. To accomplish this, the master process must coordinate the loading and unloading of slave memory. Thus, whenever a slave process loads memory, it first notifies the master process. The master checks to make sure that the pending load can complete *safely*, meaning that it will not over-subscribe the physical memory. If it cannot, the master directs blocked slaves to unload memory until the pending load will complete safely or no blocked slaves have any loaded allocations.

**Lazy Read / Lazy Write (LRLW)** For applications that exhibit part-access memory access patterns (Figure 6.1d), the previous behavior is not ideal. Under either ARAW or ARLW, each update will cause an entire allocation to be loaded, even if only a small fraction of the allocation is being updated. A better approach in this case would be to only load those parts of each allocation that are actually accessed.

The lazy read / lazy write (LRLW) strategy addresses this issue by unloading allocations using the same procedure as ARLW, but loading them at a chunk granularity rather than whole allocations. Here chunk is implementation defined, the details of which are discussed in Section 6.1.4. An ancillary advantage of a lazy read approach is that it facilitates asynchronous data transfer from disk. In aggressive reading, loading a memory allocation requires the slave process to block until all relevant parts of the allocation have been read from disk. In contrast, lazy reading performs loads at smaller granularity. This means after loading a chunk, a fast operation compared with loading an allocation, the slave process can resume execution. Meanwhile, instructed by readahead policies, the OS will continue to fetch data from the relevant file in the background. When an allocation is accessed in roughly sequential order, this translates

Figure 6.2: The state transition graph for `sbpages`. +`ONDISK` indicates that the `ONDISK` flag will be given to the `sbpage`, −`ONDISK` that it will be removed, and ±`ONDISK` it will continue to be present or absent, whichever it currently is.




to reduced time spent blocked on data transfer, since most of the data will be prefetched by the OS before it is accessed by the application.

### 6.1.4 SBMA Implementation

**Memory Acquisition and Representation** SBMA uses interposition to intercept calls to libc's standard `malloc` library. That is, a call to `malloc()` in a BDMPI program will be performed by SBMA. Our implementation of SBMA includes a parameter called the *SBMA threshold* which controls the minimum sized allocation to be managed by SBMA. Any allocation less than the SBMA threshold will be passed directly to the appropriate libc function, bypassing SBMA management. SBMA uses the `mmap()` and `munmap()` functions to acquire/release memory to/from the OS. When an application requests memory, SBMA is invoked and calls `mmap()` to obtain a virtual address from the OS pointing to a region of memory that contains the requested amount of memory. Then, a file is created on disk where the memory region will be persisted whenever it is unloaded. The memory region returned by `mmap()` is segmented conceptually into consecutive equal sized blocks, each containing a fixed number of system memory pages, referred to as `sbpages`. Each `sbpage` is associated with an `sbpage` state. Valid `sbpage` states are unmodified (`INSYNC`), modified (`DIRTY`), and not accessible (`NONE`). Furthermore, the `INSYNC` and `NONE` states can be modified with a stored on-disk (`ONDISK`) flag. The precise meanings of these states are described in subsequent sections and

Figure 6.2 provides a summary of the valid state transitions and their effect on the ONDISK flag of an sbpage. SBMA also provides the functionality to discard an allocation. This operation removes any ONDISK flags from the associated sbpages and transitions all sbpages to the NONE state. During this, no sbpages which are in the DIRTY state are written to the associated file, the allocation is simply treated as if it were newly allocated.

**Access Control and Data Persistence** In order for SBMA to perform the various read/write optimizations, it needs to know which parts of the allocations are being accessed by the application and if these accesses are read and/or write. To achieve that, SBMA uses the functionalities provided by memory protection via the Linux kernel. Each of the three states, INSYNC, DIRTY, and NONE, imply a specific set of access permissions, set using the system call mprotect(). Based on these access permissions the OS will generate the signal SIGSEGV whenever the application attempts an access that is not permitted according to the state of the corresponding sbpage. Trapping these SIGSEGV signals is one of the means by which the SBMA system transitions sbpages from one state to another; the other being an explicit discard of an allocation. To trap the SIGSEGV signal, SBMA installs its own SIGSEGV handler at the time that MPI_Init is called.

When an sbpage is in the INSYNC state, it means that its corresponding file is synchronized with its contents and implies that the application has read access to the memory corresponding to the sbpage. An sbpage in the DIRTY state indicates that the application has modified the sbpage since the last time that its file was loaded and implies read and write access permissions for the application. Both the INSYNC and DIRTY signify that the sbpage is resident in physical memory. On the other hand, the NONE state means that the sbpage is not resident in physical memory and as such, implies that there are no access permissions for the sbpage.

Initially every sbpage of a new allocation is put in the INSYNC state. When the application attempts a write access to any of these sbpages, a SIGSEGV is generated by the OS and trapped by SBMA, which transitions the sbpage to the DIRTY state and changes the access permissions to read/write. When a slave process unloads an allocation, it scans the states of its sbpages and writes to the allocation's associated

file only those sbpages which are in the DIRTY state. All sbpages in the allocation are then transitioned to the NONE state and any sbpages written to disk are given the ONDISK flag. Once an sbpage is given the ONDISK flag, it retains the flag until it is either freed or discarded. As part of unloading an allocation, the slave releases any associated physical memory resources, via the system call madvise(), invoked with the flag MADV_DONTNEED, which notifies the operating system that the specified memory resources are no longer needed. In addition, SBMA sets the permissions of the allocation to NONE, so any future access will be intercepted by SBMA and handled appropriately.

**Aggressive Read / Aggressive Write Details** In the ARAW approach, when a slave process reaches an MPI synchronization point, but before entering the blocked state, it unloads all of its memory allocations. This means that for each of its allocations, it scans for sbpages in the DIRTY state, writes them to their appropriate location in the allocation's corresponding file and updates their state to include the ONDISK flag. Afterwards, all of the allocation's sbpages are transitioned to the NONE state and the access permissions for the allocation memory are set to none. Lastly, the physical memory resources are released by the process with a call to madvise().

In an aggressive read approach, when a slave process receives a SIGSEGV, the allocation to which the offending memory location belongs is identified. Once found, the slave initiates an allocation load by reading into memory the sbpages that have the ONDISK flag, transitioning all of the sbpages to the INSYNC state and setting the access permissions to read.

**Lazy Read Details** In a lazy read approach, when a slave process receives a SIGSEGV associated with a read access, the allocation to which the offending memory location belongs is identified. Once found, the slave initiates an sbpage load on the appropriate sbpage by reading it into memory only if it has the ONDISK flag, transitioning it to the INSYNC state and setting its access permissions to read.

**Lazy Write Memory Tracking Protocol** The lazy write approach requires that the master process maintains an accurate count of the amount occupied physical memory, so that writes can be delayed. To accomplish this, messages are exchanged between the master process and relevant slave processes each time memory is loaded or unloaded.

Four different message types are relied on to provide this functionality: load, unload, release, and proceed.

When a slave process wants to load memory, it sends a *load message* to the master containing the size of the memory region to be loaded and then waits to receive a *proceed message* in response. Upon receiving a load message, the master process verifies that the requested amount of memory can be safely loaded. If not, then the master process repeatedly chooses a blocked slave process and commands it to unload all of its memory allocations via a *release message*. This is repeated until no slaves have any loaded allocations or the requested amount of memory can be loaded safely, at which point the master sends a proceed message to the slave who initiated the request. Whenever a slave successfully unloads its memory allocations due to the reception of a release message, it notifies the master with an *unloaded message* containing the aggregate amount of memory unloaded. It is not sufficient to exchange load messages and unload messages only when memory is loaded and unloaded. It is also necessary to send unloaded / load messages whenever memory is freed / allocated, since these have the same effect on the total occupied memory as unloading / loading.

When lazy writing is coupled with lazy reading, an extra consideration must be made. Rather than sending a load message each time that an `sbpage` is loaded, which would be prohibitively expensive for applications with large allocations, the LRLW approach sends exactly one load message for each allocation. The message is sent when the first access is made to any of the allocation's `sbpages`.

**Dynamically Choosing the Number of Running Processes** For some problems, the memory footprint of the slaves is not fixed throughout the lifetime of the application or cannot be easily / accurately estimated a priori. In these cases, it is desirable to have the SBMA runtime monitor available system memory and choose the number of running slaves dynamically based on the slaves' memory requirements. Choosing which slaves to allow to execute simultaneously is a hard problem due to the fact that the SBMA runtime does not know the future memory requirements of the slaves.

To address this problem, we have implemented an extension to the original SBMA runtime that allows the slaves to compete for residency, with certain constraints to ensure that the system makes progress. The basic idea is that the SBMA runtime will

allow a previously blocked slave to begin execution as long as there is enough physical memory available to satisfy its next memory request. If at any point, the memory becomes over-committed, then an ordering is imposed on the running slaves and the last slave in the order is required to evict its memory and enter the blocked state. In our implementation, the ordering is based on the amount of resident memory belonging to each slave, so that the slave with the least resident memory is subject to eviction. This simple constraint ensures that at least one slave process will always be able to make progress.

In an approach like this, there is a trade-off between the granularity at which slave processes reserve / acquire physical memory and the utilization of the available physical memory. Here physical memory utilization refers to the amount of physical memory populated by data that has been or will be accessed before a slave process reaches a pre-specified blocking point. Depending on which approach is used, either the AR or LR from Sections 6.1.4 and 6.1.4, the contention for physical memory and potential utilization will change.

In an AR scheme, the first access to an allocation will cause a slave process to reserve enough physical memory to load the entire allocation from secondary storage. Assuming that the number of allocations is relatively small, due to application design or a memory allocator which aggregates small allocations into larger ones, then the likelihood that a process will be asked to evict its memory and block before it reaches a pre-specified blocking point decreases. However, if a slave process does not access a significant portion of a particular allocation between successive blocking points, then the utilization of physical memory will be low. In this case, some other slave process may be prevented from running, due to memory constraints, despite the running slaves only actually requiring a small fraction of the memory they reserved.

In an LR approach, the first access to a memory location requires that the slave process reserve physical memory at a granularity of an `sbpage`. This approach allows more slave processes to run simultaneously and increases memory utilization when only parts of allocations are accessed between successive blocking points. However, this approach also increases the probability that a slave process will be asked to evict its memory and block before reaching a pre-specified blocking point. The reason for this is that with more slaves running simultaneously, the physical memory available to each

slave decreases.

**Multi-threaded Environments** The previous discussion addresses the needs of many distributed out-of-core applications which rely solely on MPI. However, there do exist applications which embrace the MPI+X paradigm. This is especially true of MPI+OpenMP and MPI+Pthreads. To thoroughly address the issues of SBMA in a multi-threaded environment is outside the scope of this work. However, the major challenge involved in doing so, is the race condition that occurs when one thread is loading an `sbpage` at the same time that another thread is writing to an `sbpage`. Since loading an `sbpage` requires that the destination buffer have write permissions for the duration of the load, the thread loading the `sbpage` does so. However, if another thread tries to write to the same `sbpage` during this time, two things could happen: (1) the write could be overwritten by the load or (2) the write could succeed, but SBMA would have no idea since a signal was never generated due to the write protection of the buffer. In either case, the data can be lost. This problem, known as the atomic page update problem, has been studied in other contexts, most notably with relation to distributed shared memory systems [73]. Some of the techniques described in [73] can be directly implemented within the context of SBMA and thus allow it to work for applications which rely on multi-threading within a computation node.

## 6.2 Experimental Design

**Synthetic Benchmark** The synthetic benchmark was designed to quantify the overhead associated with a memory management subsystem built on top of the Linux kernel signaling mechanisms. The benchmark allocates a single chunk of memory large enough to fill the RAM of the host machine, which in our case was slightly less than 4GB to avoid interference by the OS VMM. The synthetic benchmark is comprised of three micro-benchmarks, each executing a single type of memory access operation from the following: read (`==`), write (`=`), or read/write (`+=`). Each micro-benchmark consists of a `for` loop which iterates over the entire allocation performing the appropriate memory access operation on each byte. To increase the stability of the timings, before each micro-benchmark is executed, the system's page cache is cleared via a call to `posix_fadvise()`

and the CPU caches are cleared using unoptimized code to populate a sufficiently large segment of memory which is immediately released.

Each micro-benchmark was executed using the four possible combinations of the following memory placement policies: in-memory (I) and read (R) and memory protection policies: aggressive (A) and lazy (L). The I memory placement policy means that during the entire execution of the micro-benchmark the allocation is stored in memory, this is in contrast to the R policy where memory is stored on disk and must be read before being accessed. For a write (=) access, reading from disk is unnecessary and thus, the data is not loaded before writing. The A memory protection policy dictates that the first time that a read protection fault is raised for the allocation, the memory protection for the entire allocation is updated to read protected. The opposite of this is the L policy where each read protection fault raised causes only the corresponding `sbpage`'s memory protection to be updated to read protected. Note that this is not true for write faults, since SBMA always grants write permission exactly one `sbpage` at time.

The AI and LI combinations allow for the quantification of the inherent overhead of a memory subsystem based on the Linux kernel signaling mechanisms. The AR and LR policy combinations correspond exactly to the SBMA schemes including each of these policies. Thus, they allow us to measure the overhead which can be expected from the SBMA system. A special case, denoted as *no protection (np)* was included, in which the allocation is given read and write permission at the time of request and is not changed during execution, thus incurring no overhead. In this case, the A and L policies will only be applicable to the R memory placement policy where they dictate the resolution at which data is read from the disk. In the case of the I policy, results will be identical for A and L. This special case is representative of a BDMPI runtime without SBMA and acts as a baseline to compare the other schemes against.

For each micro-benchmark we reported the average throughput, namely, the number of system pages operated on per second. This is obtained by timing the entire `for` loop of each micro-benchmark then dividing the total time by the number of system pages in the allocation. Further, each time reported is the average throughput of ten executions of the synthetic benchmark. Due to the precautions taken, the timings results were extremely stable, $< 0.5\%$ error, thus no error statistics are reported.

**Real-world Applications** The implementation of PageRank was the same as that used in [72] and uses a one-dimensional row-wise decomposition of the sparse adjacency matrix. Each MPI process gets a consecutive set of rows such that the number of non-zeros of the sets of rows assigned to each process is balanced. Each iteration of PageRank is performed in three steps using a *push* algorithm [74].

The ParMetis implementation is version 4.0.3 and was downloaded directly from the author's website [75]. ParMetis is a graph partitioner based on the multi-level paradigm. In its initial phases of execution, the graph being partitioned is contracted into successively smaller graphs until the size of the resulting graph is small enough to directly compute a high-quality partitioning. This partitioning acts as an initial partitioning for the later phases of execution, when the graph is un-contracted and the partitioning refined. The original source code was modified for our experiments by changing a single line to disable the use of its own workspace management and thus allow SBMA to manage its memory allocations.

The SPLATT implementation is a version that was obtained directly from the author of [76] and uses a three-dimensional decomposition of the sparse adjacency tensor. Each MPI process gets a consecutive set of rows such that the number of non-zeros of the sets of rows assigned to each process is balanced. For our experiments, we compute 16 factors. Each iteration updates the factorization using the alternating least squares method and checks for convergence.

The KMeans [77] implementation uses a one-dimensional decomposition of the sparse matrix, i.e., each MPI process gets an equal number of consecutive rows. During each iteration of the algorithm, each processes assigns each of its rows to the cluster corresponding to the centroid to which it is closest. At the end of each iteration, the centroids are recomputed globally and the process is re-executed if sufficient change in centroids is observed. For these experiments, we compute 150 clusters and use random centroids for the first iteration.

The implementation of graph coloring uses the same one-dimensional row-wise as PageRank. A coloring of the graph is then computed using the Jones-Plassman algorithm [78]. During each iteration, an independent set of uncolored vertices is computed. Then the vertices of this independent set are colored in parallel. This is repeated until all vertices have been assigned a color.

Table 6.1: Dataset sizes for experiments.

| Application | Dataset | Mem (GB) |
|---|---|---|
| PageRank | uk-2007-05 | 35 |
| ParMetis | nlp-kkt240 | 13 |
| SPLATT | NELL-large | 26 |
| KMeans | RCV1 | 48 |
| Jones-Plassman | NASA | 29 |

Each row represents an application and includes the dataset used and a high-water mark for the aggregate amount of memory required by all slaves.

For the five benchmarks, we gathered results by performing five executions of each application. The times that we report correspond to the average time required to perform each iteration, which was obtained by dividing the total time by the number of iterations. As a result, the reported times include the costs associated with loading and storing the input and output data. The number of running slaves was set to one in all cases, and the total number of slaves was chosen as the smallest number of slaves such that the required memory for each slave fit completely in physical memory.

**Datasets** For the PageRank experiments the undirected version of the uk-2007-05 [79] web graph was used, with 105 million vertices and 3.3 billion edges. To ensure that the performance of the PageRank algorithm was not affected by a favorable ordering of the vertices, the vertices of the graph were renumbered randomly. For the ParMetis experiments the undirected nlpkkt240 [80] graph was used, with 28 million vertices and 760 million edges. For the SPLATT experiments the NELL-large [81] dataset was used, with 2.9 million rows, 2.14 million columns, 25.5 million fibers, and a total of 143.6 million non-zero elements. The dataset was randomly permuted to ensure the SPLATT algorithms were not affected by favorable tensor ordering. For the KMeans experiments the RCV1 dataset was used, containing over 800,000 newswire stories provided by Reuters, Ltd. for research purposes, made available by Lewis et al. [82]. The dataset was duplicated sufficiently many times to generate a dataset that would require over-subscription

of memory given a fixed number of BDMPI processes. For the Jones-Plassman experiments a tetrahedral three-dimensional mesh obtained from our colleagues at NASA was used, with 200 million vertices and 1.5 billion edges. A high-water mark for the aggregate amount of memory required by all slaves is shown in Table 6.1 for reference.

**System Configuration** The experiments were run on a dedicated cluster consisting of four Dell Optilex 9010s. Each machine is equipped with a single four-core (eight-thread) Intel Core i7-3770 @ 3.4GHz processor, 4GB of memory, and a Seagate Barracuda 7200RPM 1.0TB hard drive. Because of BDMPI's dependence on the swap-file for data storage when SBMA is disabled, the machines were set up with 300GB swap partitions. The four machines run the Ubuntu 14.04.1 LTS distribution of the GNU/Linux operating system. The C compiler used was GNU GCC 4.8.2 and the MPI implementation was MPICH.

## 6.3 Results

### 6.3.1 Synthetic Benchmark

Table 6.2 shows the throughput for the three different memory access operations: read, write, and read/write. The most relevant results with respect to the BDMPI system as a whole are the rows labeled AR and LR. When any of the columns representing `sbpage` sizes are compared against the np column, the result is a quantification of the overhead of the SBMA system. In this respect, Table 6.2 reveals two important results.

First, by comparing the throughput of the np scheme for a given memory operation to the throughput for the corresponding SBMA operation for each of the `sbpage` sizes, we derive an upper limit to the overhead associated with SBMA under the set of conditions proposed in this benchmark. Thus, we see that any given memory operation is at most 2.4 times slower under SBMA than in a system which defers memory management to the OS. For a sufficiently large `sbpage` size, the overhead introduced by the implicit signal driven memory handling of SBMA can be greatly reduced. In some cases, it can be reduced to achieve nearly the same performance as explicit memory handling. For instance, for the read (==) operation for `sbpage` size 64 using the AI or LI policy, the throughput for both policies is is less than 3% lower than the np case. Among

Table 6.2: Throughput of memory operations on the micro-benchmark.

| Operation | `sbpage` size | AI | LI | AR | LR |
|---|---|---|---|---|---|
| Read (x == y) | np | 1195 | 1195 | 28 | 30 |
| | 1 | 1195 | 537 | 28 | 30 |
| | 4 | 1194 | 927 | 28 | 30 |
| | 16 | 1194 | 1134 | 28 | 30 |
| | 64 | 1194 | 1134 | 28 | 30 |
| Write (x = y) | np | 514 | 514 | 514 | 514 |
| | 1 | 288 | 208 | 288 | 208 |
| | 4 | 373 | 325 | 373 | 325 |
| | 16 | 405 | 379 | 405 | 379 |
| | 64 | 414 | 395 | 414 | 395 |
| Read/Write (x += y) | np | 472 | 472 | 28 | 30 |
| | 1 | 276 | 198 | 28 | 30 |
| | 4 | 352 | 310 | 28 | 30 |
| | 16 | 380 | 359 | 28 | 30 |
| | 64 | 387 | 373 | 28 | 30 |

Throughput, in system pages/sec, for memory access operations under different memory management strategies using a variety of `sbpage` sizes from 1 to 64, described as multiples of the system page size, so 1 = 1 × 4096B = 4KB in our configuration. The first row for each memory access operation is the **np** scheme, where memory is obtained via a call to `mmap()` and no memory protection policies are applied to it. **A** is aggressive, **L** is lazy, **I** is in-memory, and **R** is reading. The difference between AI and AR is that in the former, data resides only in memory, while in the latter, the data is read from disk before each read access. The same is true for LI and LR.

the experiments which do not require disk access, the average difference in throughput compared to the special is approximately 1.46×.

A consequence of choosing a large `sbpage` size, not addressed by this micro-benchmark, is that for applications which do not access all bytes of an allocation, memory protection may be updated and bytes read from disk unnecessarily due to the resolution implied by the `sbpage` size. Second, by comparing the results for all of the experiments involving disk I/O, we we see that the results for all experiments, including those of the np

scheme are identical within a given policy. Thus, we conclude that the overhead related to the signal handling mechanisms inherent to SBMA are overshadowed by the disk I/O. Furthermore, since all of these results are nearly the same, even when comparing the two policies, it suggests that the disk bandwidth is the limiting factor for operations involving disk I/O.

The read (==) and read/write (+=) experiments of the AI and LI policies provide information that can be used to further quantify the overhead associated with the signal handling mechanism. By comparing the results of these experiments between A and L, we see that the L policy introduces a 17.5% overhead on average. The magnitude of the overhead for a given `sbpage` size is proportional to the `sbpage` size itself. This is as we would expect, since the L policy dictates that the memory protection of each `sbpage` must be updated independently versus the A policy which has the memory protection of the entire allocation updated at once. This overhead holds for the write(=) operation as well, since by definition, a write fault in SBMA must first generate a read fault.

### 6.3.2 Real-world Benchmarks

Each application was run on a single compute node as a serial execution, as an MPI application with the number of processes equal to the number of slaves used in the BDMPI experiments, and a standard BDMPI application, i.e., without SBMA. Although running MPI with a greater number of processes than physical processors is generally not recommended, these results are included as a sanity-check, to demonstrate that the BDMPI runtime is in fact improving performance over unaltered MPI. In the following results, entries of the form $> 300.00$ indicate that after 300 minutes, the application was still running and was terminated.

**Performance of PageRank** Table 6.3 shows the performance achieved by the different SBMA strategies on the PageRank benchmark. This benchmark does not include any results for variations of the SBMA threshold. The reason for this is that the code requires a fixed set of allocations, all of which are larger than any reasonable SBMA threshold. Thus, for PageRank, SBMA will necessarily manage all allocations.

Comparing the performance achieved by the SBMA strategies with the single node non-SBMA parallel implementations (last two rows of Table 6.3), we see that all of the

Table 6.3: PageRank runtime results (minutes).

| | #Nodes=1 | | | #Nodes=4 |
|---|---|---|---|---|
| | 4 | 16 | 64 | 64 |
| ARAW | 10.96 | 10.93 | 10.53 | 2.82 |
| ARLW | 13.75 | 13.81 | 12.55 | 3.37 |
| LRLW | 13.91 | 12.13 | 10.28 | 2.52 |
| Serial | | 14.84 | | |
| MPI | | $> 300.00$ | | 10.25 |
| BDMPI w/o SBMA | | 19.86 | | 4.34 |

Runtime results, for PageRank run on the uk-2007-05 graph. PageRank was run on a single node with 12 slaves and four nodes with three slaves per node. In both cases, the number of running slaves was one per node. The single node experiments were run with three different `sbpage` sizes, described as multiples of the system page size (4KB).

SBMA strategies lead to a decreased runtime. SBMA is 1.6×–1.8× and $> 12\times$ faster than BDMPI without SBMA and MPI respectively. For the multi-node experiments in the last column of Table 6.3, the performance improvements were similar to the single node experiments. In these cases, SBMA was 1.51× and 3.58× faster on average than BDMPI without SBMA and MPI respectively.

Surprisingly, the performance improvement over the serial application was more modest. However, as was discussed in [72], the serial version, by its very nature has a few advantages over a parallel implementation. First, since the graph is randomly permuted, distributing it to the 12 slaves is a complex operation. Second, each iteration of the computation requires an all-to-all communication. Both of these computations introduce a non-trivial amount of work into the parallel execution.

For the PageRank experiments, the performance improvement achieved by the SBMA strategies can be credited to the cooperative multi-tasking model of the BDMPI runtime. In this type of application, where nearly all of the data is accessed between blocking points, the main advantage of using a system like SBMA is that it automates the bulk load and unload before and after each blocking operation. To demonstrate this, we refer to the results from [72], where the authors performed the same PageRank experiment

Table 6.4: Amount of disk I/O for ParMetis (GB).

| | Read | | | Write | | |
|---|---|---|---|---|---|---|
| | 4 | 16 | 64 | 4 | 16 | 64 |
| ARAW | 116 | 117 | 118 | 52 | 56 | 62 |
| ARLW | 67 | 67 | 68 | 38 | 40 | 45 |
| LRLW | 25 | 28 | 32 | 38 | 40 | 45 |

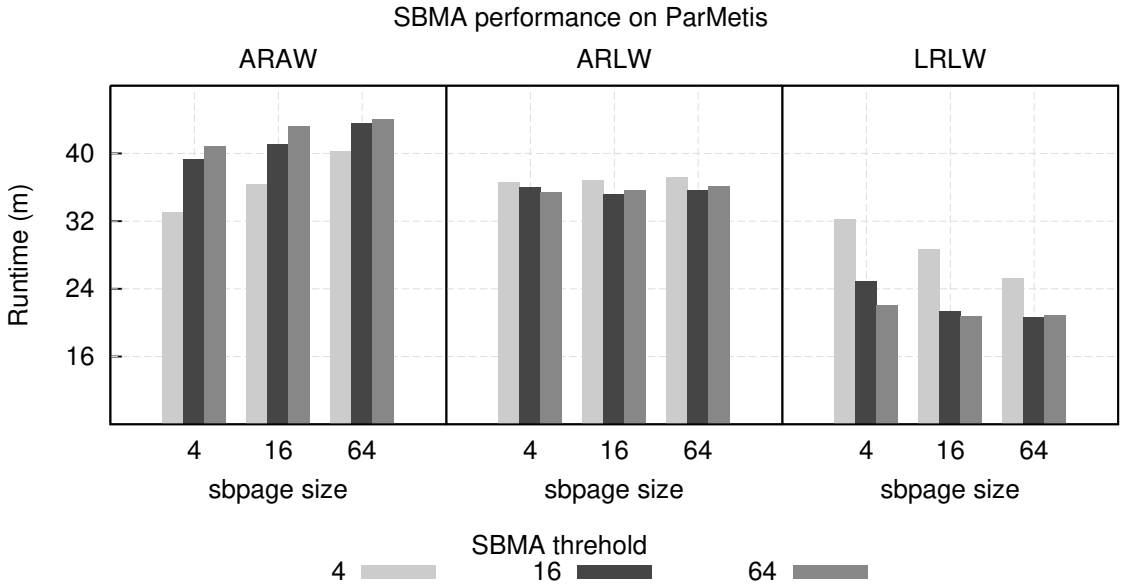
Number of GB transferred to / from disk during the execution of ParMetis on a single node with four slaves for the three SBMA strategies and three different `sbpage` sizes, described as multiples of the system page size (4KB).

using BDMPI without SBMA and a version of the PageRank application with explicit read/write to disk before/after each MPI blocking operation. The per iteration runtime for that experiment was 9.98 min for single node and 2.35 min for four node. Thus, the best of the SBMA runs were $< 5\%$ slower on average than an application optimized by hand.

Comparing only the performance of the various memory exchange strategies, we see that ARAW performs the best whereas ARLW and LRLW perform worse, but roughly the same, for all `sbpage` sizes. However, the performance difference between all three strategies is within 10%. Since each iteration of the algorithm requires that the processes access their memory allocations entirely, there is no obvious advantage of using an LR based approach. Further, since the number of slaves is chosen such that the running slave occupies most of the of available system memory, whenever a process is in the blocked state, it will be required to unload its memory, nullifying any advantage of an LW based approach. In fact the difference in runtimes between ARLW and LRLW for corresponding `sbpage` sizes is inversely proportional to the discrepancy in write operation throughput between AR and LR in Table 6.2 for the same `sbpage` sizes.

**Performance of ParMetis** Figure 6.3 shows the performance achieved by the different SBMA strategies on the ParMetis benchmark. The results for variations of the SBMA threshold were included because at various points during its execution, ParMetis allocates chunks of memory whose size is below the SBMA threshold. No multi-node experiments were run with ParMetis due to the fact that its total memory requirements

Figure 6.3: Runtime results, in minutes, for ParMetis run on the nlpkkt240 graph.




ParMetis was run on a single node with four slaves for the three SBMA strategies. Each strategy was run with three different `sbpage` sizes and three different SBMA threshold sizes, both described as multiples of the system page size (4KB). Execution of ParMetis took $> 300$, $> 300$, and 255.67 min for serial, MPI, and BDMPI w/o SBMA, respectively.

mean that it would nearly fit in the aggregate memory of the four-node system used in our experiments.

The results for ParMetis for all SBMA strategies show a large performance improvement over the non-SBMA implementations found in Figure 6.3. In this case, all three of the SBMA strategies outperformed BDMPI without SBMA with speedup ranging from 7× to 12×. The results compared with the serial execution, are even better, where SBMA achieved speedup $> 20\times$. The poor performance of serial ParMetis and non-SBMA runs can be attributed to the reliance on the OS VMM swapping mechanism and the random memory access patterns of ParMetis during the execution of its first and last few stages. During these stages, the graph being contracted or un-contracted is within a small factor of the size of the original graph, thus requiring more memory than physically available.

Comparing the performance achieved by the various memory exchange strategies, we see that LRLW performs the best whereas ARAW performs the worst, with ARLW in between. The runtimes of ARLW are 10% less than ARAW on average and those of LRLW are 33% less than ARAW on average. This increased gap in performance is expected for an application like ParMetis, which is an example of the multi-level paradigm. In its initial phases of execution, the graph being partitioned is contracted into successively smaller graphs, thus each successive phase requires a lesser amount of memory be loaded than the previous phase. This explains the performance improvement of ARLW over ARAW, since during the later phases of contraction, the memory of all slave processes can fit in the available system memory. This fact makes the loading and unloading of entire allocations, as in ARAW, unnecessary. Then, in the later phases of execution, the graph is un-contracted and refined. In most cases, although the un-contracted graph will grow from phase to phase, the amount of memory accessed will remain small relative to the size of the un-contracted graph because only the information associated with the interface vertices is accessed. Since only a relatively small amount of memory is being loaded during each of the un-contraction phases, ARAW and ARLW both suffer from excessive loading and unloading of memory, due to the transfer of data to and from disk at the allocation resolution. Table 6.4 presents the number of GB transferred to/from disk and shows that although LRLW writes roughly the same amount of data as ARAW and ARLW, it reads 75% less data than ARAW and 57% less than ARLW. The characteristics of these two phases, contraction and un-contraction, make LRLW ideally suited for an application like ParMetis, which is supported by the results.

Looking at just the effect of `sbpage` size on the three strategies we see that each is affected differently. ARAW is affected adversely, ARLW neutral, and LRLW positively. To understand this, we have to consider the implications of using different `sbpage` sizes. Under all three strategies, the consequence of a larger `sbpage` size is that the number of bytes included in `DIRTY sbpages` can be unnecessarily increased if the application is not writing to all bytes within an `sbpage`. This is very likely to happen during later contraction and un-contraction phases, when the memory access pattern becomes more sparse. As Table 6.4 shows, in all cases, as `sbpage` size increases, so does disk I/O. Since this is the only effect of `sbpage` size on ARAW we see that ARAW's runtime increases

Table 6.5: SPLATT runtime results (minutes).

| | #Nodes=1 | | | #Nodes=4 |
|---|---|---|---|---|
| | 4 | 16 | 64 | 64 |
| ARAW | 12.01 | 13.04 | 13.47 | 2.23 |
| ARLW | 11.46 | 12.28 | 11.72 | 2.13 |
| LRLW | 11.92 | 8.41 | 7.17 | 2.57 |
| Serial | | 13.52 | | |
| MPI | | $> 300.00$ | | 5.38 |
| BDMPI w/o SBMA | | 38.32 | | 4.01 |

Runtime results, for SPLATT run on the NELL-large graph. SPLATT was run on a single node with eight slaves and four nodes with two slaves per node. In both cases, the number of running slaves was one per node. The single node experiments were run with three different `sbpage` sizes, described as multiples of the system page size (4KB).

as the `sbpage` size increases. In ARLW, the increased disk I/O is offset by the nature of lazy writing and its expression in ParMetis. Many of the ParMetis iterations which are likely to cause unnecessary disk I/O, due to sparse memory accesses, are also likely to be candidates to benefit from a lazy write strategy. Like ARLW, LRLW offsets the effects of sparse memory accesses using lazy writing, but has one additional advantage. As `sbpage` size increases, the memory protection overhead related to the lazy reading strategy decreases. This effect of `sbpage` size was confirmed in Table 6.2. Thus, LRLW realizes a new gain in performance as `sbpage` size increases as can be seen in Figure 6.3.

**Performance of SPLATT** Table 6.5 shows the performance achieved by SPLATT for each of the three SBMA strategies. For the same reasons as the PageRank benchmark, results for variations of the SBMA threshold were omitted.

For the single node experiments, the characteristics of the SPLATT experiments were similar to those of the ParMetis experiments. Namely that the addition of SBMA improved performance in all cases when compared with the serial version and BDMPI without SBMA. Also, despite the additional overhead of lazy reading and writing, performance can be improved by enabling these optimizations. Like the ParMetis results,

Table 6.6: Amount of disk I/O for SPLATT (GB).

| | Read | | | Write | | |
|---|---|---|---|---|---|---|
| | 4 | 16 | 64 | 4 | 16 | 64 |
| ARAW | 70 | 70 | 70 | 24 | 24 | 24 |
| ARLW | 51 | 51 | 51 | 20 | 20 | 20 |
| LRLW | 23 | 23 | 23 | 20 | 20 | 20 |

Number of GB transferred to / from disk during the execution of SPLATT on a single node with eight slaves for the three SBMA strategies and three different `sbpage` sizes, described as multiples of the system page size (4KB).

the performance improvement between the three schemes can be explained most easily by referring to Table 6.6, which shows that the amount of disk I/O for SPLATT. Like ParMetis, ARAW was the slowest and also had the highest amount of disk I/O and LRLW was the fastest and had the lowest amount of disk I/O.

For the multi-node experiments, the results are generally as expected. Each of the three strategies performed had runtimes 3×–6× faster than their single node counterpart. As anticipated, ARLW was faster than ARAW. However, not as expected was the performance of LRLW compared with the other strategies. As it turns out, the cumulative memory on the four nodes was large enough to support a great deal of deferred writes. Thus, the relatively low aggregate disk I/O, combined with the overhead of lazy reading, actually had a negative impact in the LRLW scheme, causing its runtime to be larger than both ARLW and ARAW. For reference, the amount of data written to disk was 24GB, 14GB, and 20GB for ARAW, ARLW, and ARLW respectively.

**Performance of KMeans** Table 6.7 shows the performance achieved by KMeans for each of the three SBMA strategies. For the same reasons as previous problems, these results do not include any variations of the SBMA threshold.

For the single node experiments, SBMA resulted in runtimes that were 1.5×–1.8× and $> 4\times$ faster than BDMPI without SBMA and MPI, respectively. The results compared to the serial implementation were 2.3× faster on average. This improvement over the serial implementation is due to the data locality benefits of decomposing the problem, a step required for the parallel execution, since each process is responsible for

Table 6.7: KMeans runtime results (in minutes).

| | #Nodes=1 | | | #Nodes=4 |
|---|---|---|---|---|
| | 4 | 16 | 64 | 64 |
| ARAW | 111.32 | 110.75 | 109.02 | 39.38 |
| ARLW | 98.09 | 97.64 | 97.10 | 36.01 |
| LRLW | 101.58 | 99.97 | 98.38 | 37.62 |
| Serial | | 238.42 | | |
| MPI | | $> 300.00$ | | 94.87 |
| BDMPI w/o SBMA | | 169.32 | | 59.36 |

Runtime results, for KMeans run on the RCV1 dataset. KMeans was run on a single node with 16 slaves and four nodes with four slaves per node. In both cases, the number of running slaves was one per node. The single node experiments were run with three different `sbpage` sizes, described as multiples of the system page size (4KB).

computing only a subset of the rows of the matrix. In the serial implementation no such decomposition is performed. Thus, each iteration requires multiple passes over the dataset with no data reuse. For the multi-node experiments, the performance improvements for SBMA were similar to those observed on a single node. Namely that the addition of SBMA improved performance in all cases when compared with the results of the baseline methods.

The results for the three SBMA strategies are similar to the those of PageRank, with one notable difference. For KMeans, the ARAW was the worst performing SBMA strategy instead of the best. This can be explained by the following characteristic of the KMeans algorithm. In KMeans, the dataset is accessed multiple times during each iteration and these accesses are separated by a global reduction of the centroid values. Since the number of clusters being computed is small relative to the size of the matrix, very little data is needed by each process to compute the reduction. Under a lazy write strategy, this means that the last process to reach the reduction point can keep its memory resident and begin executing after the reduction has completed without the need to load any memory. The result of this is that ARAW reads a total of 227GB of data from disk during its execution, compared with 221GB and 220GB for ARLW

Table 6.8: Jones-Plassman runtime results (minutes).

| | #Nodes=1 | | | #Nodes=4 |
|---|---|---|---|---|
| | 4 | 16 | 64 | 64 |
| ARAW | 12.91 | 12.42 | 12.37 | 4.42 |
| ARLW | 12.78 | 12.45 | 12.42 | 4.07 |
| LRLW | 11.62 | 11.51 | 11.48 | 3.59 |
| Serial | | 27.35 | | |
| MPI | | $> 300.00$ | | 8.10 |
| BDMPI w/o SBMA | | 32.33 | | 10.59 |

Runtime results, for Jones-Plassman run on the NASA dataset. Jones-Plassman was run on a single node with eight slaves and four nodes with two slaves per node. In both cases, the number of running slaves was one per node. The single node experiments were run with three different `sbpage` sizes, described as multiples of the system page size (4KB).

and LRLW, respectively. This explains the more than 10% slower runtimes for ARAW compared with both ARLW and LRLW. The slight improvement of ARLW compared to LRLW can be contributed to the overhead associated with managing lazy reading when the dataset is accessed in its entirety between blocking points.

**Performance of Jones-Plassman** Table 6.8 shows the performance achieved by Jones-Plassman for each of the three SBMA strategies. For the same reasons as previous problems, these results do not include any variations of the SBMA threshold.

For the single node experiments, the characteristics of the Jones-Plassman compared to the baseline methods were similar to those of the ParMetis and KMeans experiments. Again, the addition of SBMA improved performance in all cases when compared with the serial version, MPI, and BDMPI without SBMA. The runtimes of Jones-Plassman with SBMA were between 2× and 3× faster than the runtimes of the serial and non-SBMA runs respectively and significantly faster compared with MPI. For the multi-node experiments, the results are generally as expected. Each of the three strategies performed had runtimes approximately 3× faster than their single node counterpart. As anticipated, LRLW was the fastest method and ARAW the slowest, with ARLW in

Table 6.9: Runtime results for dynamically choosing number of running slaves (minutes).

| | ParMetis | | | SPLATT | | |
|---|---|---|---|---|---|---|
| # Running | ARAW | ARLW | LRLW | ARAW | ARLW | LRLW |
| 1 | 25 | 20 | 22 | 11 | 11 | 10 |
| ∗ | 23 | 19 | 11 | 6 | 6 | 6 |

Runtime results, for allowing the SBMA runtime to dynamically choose the number of running slaves for the ParMetis and SPLATT benchmarks. Each problem was run with the three SBMA variations with one running slave, indicated by the # Running 1 row, and with dynamically chosen number of running slaves, indicated by the ∗ row.

between.

Comparing only the performance of the various memory exchange strategies, as shown in Table 6.8, we see that LRLW performs the best whereas ARAW and ARLW perform worse, but roughly the same, for all `sbpage` sizes. Like PageRank, the performance difference between all three strategies is within 10%. From the results of ARAW and ARLW, we can conclude that lazy writing has little impact on the performance of Jones-Plassman. This make sense, since like PageRank, the algorithm makes a single pass over the entire dataset each iteration and more importantly between each blocking point. Thus, after reaching a blocking point, it is necessarily the case that a process will evict its memory to make room for the next running process. However, unlike PageRank, which traverses the graph and unconditionally updates the PageRank vector for each vertex, in the Jones-Plassman algorithm, vertex data is not unconditionally updated. Rather, during each iteration of the algorithm, vertex data is updated based on the state of neighboring vertex data. This conditional updating of data gives an advantage to LR based approaches, since they do not read entire allocations upon first access. A consequence of this is the roughly 10% reduction in runtime that we see between LRLW and the two alternative memory exchange strategies.

**Impact of Dynamically Choosing the Number of Running Slaves** Table 6.9 shows the performance achieved by the BDMPI runtime on the ParMetis and SPLATT benchmarks. These two benchmarks were chosen because they represent applications where the memory footprint of the BDMPI slaves changes throughout the lifetime of

the application. In contrast to the previous experiments, which were all run using a Seagate Barracuda 7200RPM 1.0TB, these experiments were run on a Samsung SSD 850 PRO 256GB. The reason for this is that the previous experiments were limited to one running slave per node, so there was no contention for secondary storage. However, for these experiments, the number of running slaves is necessarily greater than one. For this reason, we chose to perform the experiments on an SSD for improved performance when multiple slaves access secondary storage.

Comparing the performance of letting the runtime dynamically change the number of running slaves and using a single running slave, we see that for the two benchmarks, dynamically choosing the number of running slaves decreases runtime. Looking specifically at ParMetis, we see that performance improvement is moderate for ARAW and ARLW, but 2× better for LRLW. The marginal improvement in ARAW and ARLW is due to their reliance on the AR approach. As mentioned in Section 6.3.2, during un-contraction in ParMetis, only a relatively small portion of each allocation is actually accessed by the slave processes for computation. In the ARAW and ARLW approaches, this translates to a smaller number of slave processes which can be running simultaneously, since each running slave, despite requiring a small memory footprint, must reserve a large amount of physical memory. This is in contrast to the LRLW approach which only reserves the exact amount of memory needed, within a factor of an `sbpage`.

For the SPLATT benchmark, the performance results are quite different. As Table 6.9 shows, the performance of all three SBMA variations with dynamically chosen number of running slaves is roughly the same, and is approximately 2× better than SBMA with a single running slave process. These two results are is explained by a few observations related the memory access patterns of SPLATT. Like ParMetis, the memory footprint of slave processes in SPLATT varies throughout its execution. However, unlike ParMetis, the variation is regular, i.e., during each iteration of the algorithm, there is one step which requires a large memory allocation to be resident in memory, then a step that requires only a small amount of memory to be resident. Recall that the total number of slaves is chosen so that the high-water mark for memory residency for any slave will be less than the available physical memory. Thus, the first step in SPLATT, which requires large memory residency dictates the total number of slaves, despite the fact that during the second step, it is conceivable that all slaves can be

resident in memory and computing simultaneously. As a result, with one running slave, the second step of SPLATT is unnecessarily serialized within a node. This explains the improvement realized by dynamically choosing the number of running slaves. The consistency of runtimes for the three SBMA variations is due to the fact that during the second step of SPLATT, the allocations accessed are accessed in their entirety. Thus, the choice of AR or LR has little impact, since the will need to be totally resident. Since, the size of these allocations is small, the allocations required by multiple slaves will safely fit in physical memory, meaning that the eviction policy, AW or LW, also has little impact.

Finally, when Table 6.9 is compared with Figure 6.3 and Table 6.5, we see that for a single running slave per node, the runtimes for HDDs are within a factor of two of their SSD counterparts. This is a good improvement considering the bandwidth of the HDDs used in the experiments was roughly a factor of five less than that of the SSDs. This improvement can be attributed to the reduced contention on the secondary storage facilitated by the node-level co-operative multi-tasking execution model of BDMPI coupled with the bulk I/O performed by SBMA, especially when writing.

# Chapter 7

# Shared Memory Architecture

## 7.1 Methods

### 7.1.1 OpenOOC: Open Out-of-Core

Data-parallel problems [83] are an exemplary use case for shared-memory parallel programming languages / language extensions. By their nature, shared memory data parallel problems often exhibit a large amount of concurrency. There has been a significant amount of research, performed in a variety of contexts, related to exploiting this concurrency to hide latency. A few well-known examples of this in hardware are GPUs [84, 85] and the Tera computer system [86]. In software, Distributed Shared Memory (DSM) systems exploit this concurrency to hide the latency of remote memory accesses [87–89]. In any of these systems, the basic premise is the same. When an execution context, be that in hardware or software, encounters a long latency operation, the system switches to a different execution context and performs useful computations there until that context reaches a long latency operation or a pending operation completes. However, to the best of our knowledge, there has been no work that applies this idea to shared memory out-of-core computations. In many ways, this idea is very similar to the execution model of SBMA. Recall that the SBMA runtime exploits additional concurrency provided by an over-decomposition of the problem, to hide the latency associated with exchanging data between physical memory and secondary storage. In the case of shared memory, the additional concurrency required for hiding latency of out-of-core computation can

be exposed through any of the data-parallel shared memory programming models.

In this chapter we investigate the extension of the memory control mechanisms implemented in SBMA to a shared memory runtime. We use OpenMP [90] as the data-parallel shared memory programming model to provide the runtime with the required concurrency. This work does not extend the OpenMP standard or its runtime system. Rather it uses OpenMP in a very limited capacity, and only as a way to express concurrency. It does not currently consider OpenMP directives, i.e., `FLUSH`, that affect that basic memory model of OpenMP [91]. The only directive considered in this work is the `PARALLEL FOR` worksharing directive, although the following discussion is relevant to any worksharing directives with similar semantics. The OpenMP standard [1] specifies that the iterations of a `for`-loop adorned with the `PARALLEL FOR` directive can be executed in parallel. Based on the discussion in Section 5.1.2, this implies that the iterations can also be executed concurrently.

We call the new extension *Open Out-of-Core* (OpenOOC). OpenOOC is implemented as an application layer execution scheduler and is intended to be used transparently with those programs written in OpenMP that have the need to execute in external memory. In this case, instead of having multiple OS processes available for execution at a time, like in SBMA, the runtime will maintain multiple user space execution contexts. Whenever an exchange of data between physical memory and secondary storage is required, the runtime can initiate an asynchronous exchange of the data and switch execution contexts to continue performing useful computation while the exchange completes. In some ways this is similar to what would happen if threads were over provisioned in a standard OpenMP environment, where the OS could switch between threads in such a situation. However, since the OS relies on preemptive scheduling, in the case that threads are over provisioned, the system will likely suffer from resource contention. Instead, OpenOOC leverages cooperative multitasking, as in SBMA, to effectively hide the latency incurred by data exchange, so long as the rate that the runtime can switch execution contexts is much higher than the time required for the exchange of data.

### 7.1.2 OpenOOC Architecture

To accomplish this, the OpenOOC runtime is situated alongside the OpenMP runtime. Whenever program execution reaches an OpenMP `PARALLEL FOR` directive and

the OpenMP runtime creates a group of threads, the OpenOOC runtime creates a corresponding group of user space execution contexts, henceforth called fibers, for each OpenMP thread. Fibers are the mechanism by which a single thread concurrently executes its assigned loop iterations. This is achieved by treating a thread's set of loop iterations as a work queue for its corresponding fibers. In effect, the execution of a thread's loop iterations becomes a sequence of switching between fibers, until the work queue has been exhausted. Although the threads themselves are preemptively scheduled by the OS, within a thread, fibers are cooperatively scheduled. In other words, once a thread has begun executing a fiber, it runs the fiber until it encounters a non-resident data access or until there are no more loop iterations available in the work queue. If a fiber attempts to access data that is not resident in physical memory, then the OpenOOC runtime issues an asynchronous swap of the data on behalf of the thread, and the thread saves the fiber's execution context and switches to another fiber, whose data is resident in memory. If no such fiber exists, then the thread waits until one does, i.e., an asynchronous swap completes. Once a thread has completed all of its iterations, its control is returned to the OpenMP runtime so that it can be joined with the other threads.

To better understand the mechanics of the OpenOOC architecture, it is helpful to walk through a concrete example. For this, we will use a dense matrix multiplication function written in C, seen in Listing 7.1. Not only will this example help illustrate the fundamentals of the OpenOOC architecture, but it also serves as the computational in Section 7.3. First, recall that if this example were compiled without OpenMP support, then it would behave exactly as if line 8 had been omitted, i.e., an unoptimized C implementation of dense matrix multiplication. However, with OpenMP support, this function is compiled so that when execution reaches the `PARALLEL FOR` directive (line 8), the OpenMP runtime creates a group of threads for parallel execution[1]. The exact number of threads could have been specified using the `NUM_THREADS` clause, set using an environment variable, or via an OpenMP library call. Likewise, the distribution of iterations amongst threads could have been specified by a `SCHEDULE` clause, an environment variable, or as a runtime library call.

[1] In some implementations, the OpenMP runtime maintains a pool of threads and allocates threads from this pool to avoid the overhead of OS thread creation.

Listing 7.1: An OpenMP implementation of dense matrix multiplication in C.

```
1  double matmult(
2    int m, int n, int p, // dimensions of the matrices
3    double ** A,         // A is n × m
4    double ** B,         // B is m × p
5    double ** C          // C is n × p
6  )
7  {
8    #pragma omp parallel for
9    for (int i=0; i<n; ++i) {
10     for (int j=0; j<p; ++j) {
11       for (int k=0; k<m; ++k) {
12         C[i][j] += A[i][k] * B[k][j]
13       }
14     }
15   }
16 }
```

In this example, each iteration of the outermost `for`-loop (line 9) computes a single vector dot-product of one row from matrix `A` and one column from matrix `B`, and stores the result in a single entry of matrix `C`. There are no data dependencies between iterations and thus, each can be computed concurrently with every other. Consider the case in which the number of threads is specified to be four prior to the invocation of this function. Then each of the threads is responsible for computing `n`/4 iterations of the outermost `for`-loop. Assuming that there are at least four cores in the CPU and that there is sufficient memory in the machine for all three matrices, `A`, `B`, and `C`, then the calculation can be performed completely in parallel with no contention for resources.

Now consider the case when the amount of memory in the machine is insufficient. For simplicity, assume the memory available is enough to hold four rows of `A`, the entire matrix `B`, and four rows of `C`. In other words, there is enough memory for each thread to perform a single iteration of the outermost `for`-loop, but no more. In this case, after each thread completes its first iteration, every iteration thereafter will require exchanging data between memory and disk. Namely, the data for the row from `A` and

the row from `C` will need to be swapped into memory; all of the data from matrix `B` will remain in memory for the lifetime of the computation. If data is exchanged with disk at a granularity of $\beta$ datums, then every iteration will require the thread to swap $(m+p)/\beta$ times. Each time that a swap is required, the thread will stop doing useful work and wait while the data is moved. If the time it takes to move $\beta$ datums is expressed as $\alpha$, then the total time spent not doing useful work by a single thread is $n\alpha(m+p)/4\beta$.

OpenOOC addresses this by having the thread switch to do other useful work while the data is moved. The other useful work in this example is performing calculations for another row of the matrices `A` and `C`. In this example, it is possible for a thread to work on $m/2\beta$ different rows concurrently, without exhausting the resources. Doing so will not reduce the total number of swaps, only the time spent idle by a thread. If the time that it takes to switch from computing one row to another is expressed by $\sigma$, then the total time spent not doing useful work is $n\sigma(m+p)/4\beta$. So, as long as $\sigma \ll \alpha$, then the computation will complete faster.

### 7.1.3 OpenOOC Implementation

OpenOOC leverages a functionality provided by the Linux kernel, known as user contexts, to implement the per thread fibers. A user context stores execution state, including all registers and CPU flags, the instruction pointer, and the stack pointer, as well as the stack itself. This provides the OS with all of the information needed to depart from the standard fetch-decode-execute instruction cycle and execute code at arbitrary locations in memory.

To operate alongside the OpenMP runtime, the OpenOOC system currently requires some minor modifications to the usage of OpenMP described in the previous subsection. Instead of using the shorthand `PARALLEL FOR` directive, it is necessary to separate the directives into two embedded code blocks, i.e., a `FOR` block inside of a `PARALLEL` block. The reason for this is that OpenOOC needs to perform setup and teardown before and after the `for`-loop inside of the parallel region. It would be possible to use some preprocessing to feign the transparency of this process, but since this is initially a work of academic interest we opted for a simple, and more explicit approach.

To initialize the OpenOOC runtime and create the fibers belonging to each thread, we introduced a library call, `ooc_set_num_fibers()`, which must be the first statement

in a code block adorned with a `PARALLEL` directive. To destroy the fibers when the parallel computation has finished, there is a corresponding library call, `ooc_finalize()`. Initially, all of the fibers are considered to be in an idle state. When execution reaches the `FOR` directive enclosed in the `PARALLEL` block, the first fiber is activated and begins executing the first iteration of the `for`-loop assigned to its parent thread. The fiber executes, uninterrupted, until it attempts a non-resident memory access, see Non-resident Memory Access below. When this happens, the fiber is put into a waiting state, and program control returns to the parent thread, as if the iteration had completed. At that point, the OpenOOC scheduler is invoked to schedule another fiber.

**Fiber Scheduling** In order to schedule the fibers, the body of the `for`-loop is encapsulated in a function which is then passed to the fiber scheduling function, `ooc_schedule()`. This loop function will receive any necessary variables as input parameters, see Listing 7.2, which is the OpenOOC equivalent of Listing 7.1.

Since each iteration of the OpenMP `for`-loop will be executed by a single fiber, the scheduling function is responsible for determining which fiber to execute next. The scheduling function is only invoked when the currently executing fiber reaches a non-resident memory access or completes its work. Thus, the scheduling function must decide between switching to a fiber waiting on a non-resident memory access, but whose memory is now resident, and starting the execution of a new fiber, i.e., another iteration of the OpenMP `for`-loop. In its current implementation, the scheduler will always prefer to simply switch to an idle fiber if there is more work to do, even if there is an outstanding fiber that is now ready to execute.

To switch from one fiber to another, the scheduling function stores the user context for the currently executing fiber and switches to the user context for the desired fiber to begin or resume execution. In Linux, this can be accomplished in a variety of ways, using the `setjmp()` and `longjmp()` functions, the `sigsetjmp()` and `siglongjmp()` functions, or the `makecontext()` and `swapcontext()` functions. The choice of which pair of functions is best for switching contexts is dependent on the hardware and software environment that the runtime will be executed in.

Listing 7.2: An OpenOOC implementation of dense matrix multiplication in C.

```
1  void loop_kern(
2    int i,         // the loop variable
3    int m, int p, // dimensions of the matrices
4    double ** A,  // input matrix
5    double ** B,  // input matrix
6    double ** C   // output matrix
7  )
8  {
9    for (int j=0; j<p; ++j) {
10     for (int k=0; k<m; ++k) {
11       C[i][j] += A[i][k] * B[k][j]
12     }
13   }
14 }
15
16 double matmult(
17   int m, int n, int p, // dimensions of the matrices
18   double ** A,          // A is n × m
19   double ** B,          // B is m × p
20   double ** C           // C is n × p
21 )
22 {
23   #pragma omp parallel
24   {
25     ooc_set_num_fibers(NUM_FIBERS);
26
27     #pragma omp for
28     for (int i=0; i<n; ++i) {
29       ooc_sched(loop_kern);
30     }
31
32     ooc_finalize();
33   }
34 }
```

**Non-resident Memory Access** In order to detect non-resident memory accesses, OpenOOC relies on the same set of memory protection mechanisms as used by the SBMA virtual memory subsystem, see Section 6.1.4. Namely, the `mmap()` and `mprotect()` system calls are used to allocate and monitor access to memory. Like SBMA, memory that is non-resident is given access permissions that will cause a `SIGSEGV` to be raised should the process attempt to access the memory. After the data has been successfully swapped into memory, i.e., been made resident, the required access permissions are granted, so that no further `SIGSEGV`s are raised, until a future time when the memory becomes non-resident. However, unlike SBMA, the OpenOOC system not only detects and swaps data in response to non-resident memory accesses, it also switches user contexts.

## 7.2 Experimental Design

**Synthetic Benchmark** The synthetic benchmark was designed to quantify the performance cost associated with the OS memory management subsystem as well as switching user contexts. The synthetic benchmark is suite of micro-benchmarks, each of which exercises a different capability of the host system. The first micro-benchmark is designed to measure the latency of swapping a page of memory via the OS memory management subsystem. To do this, the benchmark fills the RAM of the host machine. To ensure that the benchmark program can control when a page of memory will be swapped into RAM, it is necessary to manage the OS kernel functionality known as *read-ahead*. The read-ahead functionality is the means by which the OS pro-actively fetches data on behalf of a program. The aggressiveness of the OS's prefetching mecahnism is controlled by an OS kernel setting known as *page-cluster*. If a program requests a page that is not in memory, the OS will swap the page in a cluster of $2^{page-cluster}$ system pages. Thus, in the benchmark program, the page-cluster setting is a tunable parameter. The benchmark then allocates a block of memory and populates it with random data. A `for`-loop is used to iterate and access some number of datums from the allocated block of memory. To force a swap operation for each data access, the benchmark is designed to choose data access that will not overlap memory pages of previously accessed data. The second micro-benchmark measures the time for a `SIGSEGV` to be raised a memory

access permission fault. This benchmark allocates a block of memory with no access permission. Then, starting from the beginning of the block of memory, it visits consecutive pages. Each page that is accessed will cause a `SIGSEGV` to be raised. After catching the signal, the user context is manipulated so that after returning from the signal handler the next instruction to be executed will be to access the next page in the block of memory. Manipulating the user context is just a matter of manipulating the instruction pointer in the current context, which incurs almost no cost at all. The time required to visit all pages is timed and then averaged over the number of pages in the block of memory. The third benchmark is designed to measure the time required to switch from one user context to another. The program consists of a single `for`-loop that switches user context each iteration, using the `swapcontext()` system call.

For each micro-benchmark, we reported the average throughput, namely, the number of operations, swaps, signals, or context switches, per second. This is obtained by timing the entire `for`-loop of each micro-benchmark, then dividing the total time by the total number of operations. Further, each time reported, is the average throughput of ten executions of the synthetic benchmarks. Due to the precautions taken, the timing results were extremely stable, $< 0.5\%$ error, thus no error statistics are reported.

**Computational Benchmarks** The computational benchmark is the matrix-multiplication program described in Sections 7.1.2 and 7.1.3. This computational benchmark was chosen because of its predictable memory access patterns. Because of this, it is possible to study and characterize the performance of the OpenOOC system under different system configurations. The only notable difference between the benchmark used in the following section and that described in the previous sections is that the implementation used for experiments employs tiling. The tiling algorithm implemented in the benchmark is the matrix-multiplication tiling algorithm described in [16], which is also the optimal algorithm for out-of-core matrix-multiplication in terms of I/O. In order to expose enough concurrency to benefit from the OpenOOC approach, our implementation concurrently computes the partial dot-products within a tile.

The size of the matrices was chosen such that the memory required for the computation of a single tile fills the available memory. Thus, the fiber can execute concurrently without exhausting resources, but each tile will require swapping memory.

Table 7.1: Throughput of system operation on the micro-benchmarks.

| | page-cluster | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| Swap | 22610 | 14406 | 8969 | 5336 | 2864 | 1791 |
| `SIGSEGV` | 1344086 | | | | | |
| `swapcontext` | 4291845 | | | | | |

Throughput, in swap operations per second, for memory access operations under different page-cluster sizes, described as powers of two of the system page size (consistent with Linux kernel documentation), so $1 = 2^1 \times 4096\text{B} = 8\text{KB}$ in our configuration. The second part of the table shows throughput in system operations per second, for relevant system calls.

This is a realistic assumption, since the number of fibers could be adjusted to ensure this in a real-world problem. In order to reduce the computational time of the matrix-multiplication benchmark, we opted to make 3GB of the systems 4GB unavailable to the benchmark application. The dimensions of a single tile was computed as a function of the OS parameter known as *page-cluster*. The page-cluster parameter determines the granularity at which the OS swaps pages between memory and disk. Thus, tile dimensions were adjusted so that all computations associated with a tile could be carried out concurrently without competing for resources.

**System Configuration** The experiments were run on a dedicated Dell Optilex 9010 system. The machine is equipped with a single four-core (eight-thread) Intel Core i7-3770 @ 3.4GHz processor, 4GB of memory, and a Samsung SSD 850 PRO 256GB hard drive. Because of OpenOOC's dependence on the swap-file for data storage, the machine was set up with 64GB swap partition. The machine runs the Ubuntu 14.04.1 LTS distribution of the GNU/Linux operating system. The C compiler used was GNU GCC 4.8.2.

## 7.3 Results

### 7.3.1 System Benchmark

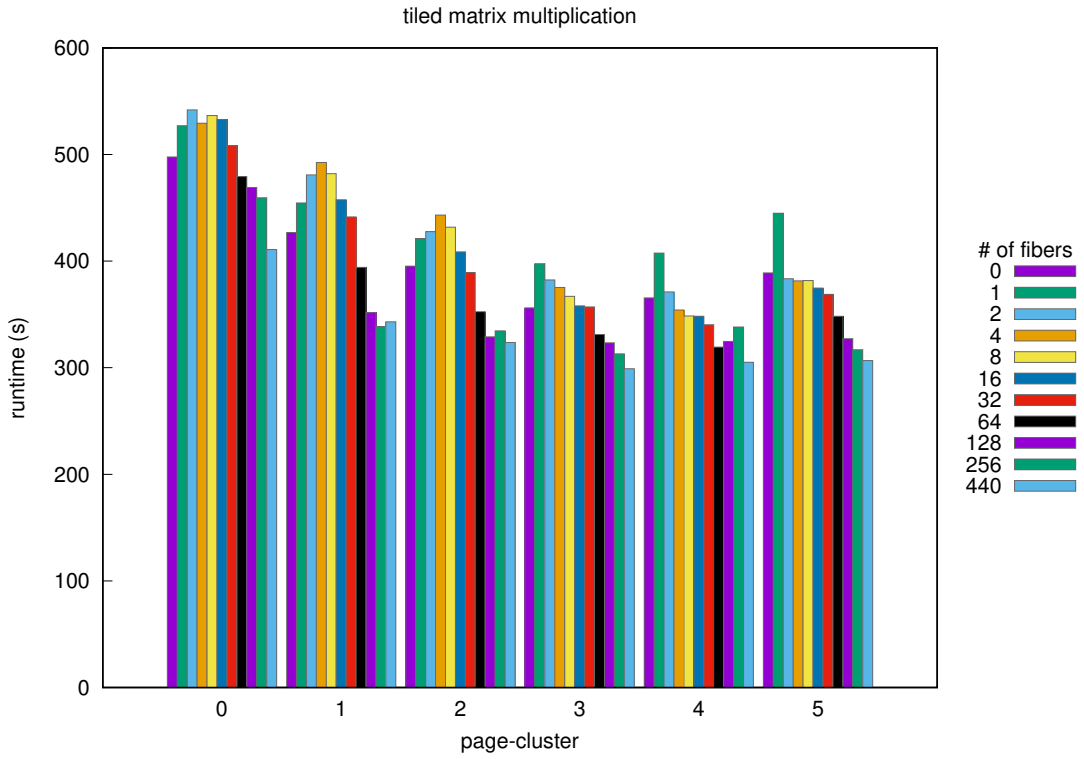
Table 7.1 shows the throughput for system operations relevant to the OpenOOC system. In the table, swap throughput is measured in operations per second rather than system pages per second because the latter is only useful if the problem exhibits spatial locality. In this analysis, we are more interested in the general performance characteristics of the system, rather than optimizations designed for particular access patterns. The significance of this table is to quantify the overhead associated with the user space system capabilities required by OpenOOC, and compare this overhead to the costs associated with no intervention, i.e., allowing the OS to manage everything. In this respect, Table 7.1 reveals an important result.

The result is that the overheads introduced by OpenOOC's execution model, where non-resident memory accesses cause the system to switch execution contexts, are small relative to the cost of swapping a page of memory. If we look at the two system calls required to detect a non-resident memory access individually, each of their throughputs is significantly higher than the throughput of an OS swap operation. In fact, each of them by themselves has an average throughput at least 60× greater than swapping a single page of memory. This factor goes up as the *page-cluster* is increased. When considered in a configuration that one would find in the OpenOOC runtime, a `SIGSEGV` followed by a context switch to the scheduler followed by a context switch to the newly executing fiber, we see that the throughput is still more than 30× greater. This does not imply that the system overall will be 30× faster, since there is other logic required to managed the OpenOOC runtime system which is not measure in this table. This table is just a measurement of the performance of the individual system calls, relative to the swap operation, which serves as a good upper-bound for performance.

### 7.3.2 Computational Benchmarks

Figure 7.1 shows the runtime for multiplying two matrices under a variety of values for the OS page-cluster parameter. In this figure, the left-most bar for each page-cluster group represents the baseline out-of-core matrix-multiplication algorithm. For each of

Figure 7.1: Runtime results, in seconds, for tiled matrix-multiplication with and without OpenOOC.




the page-cluster values, the tile size was adjusted so as not to violate the requirement that all memory necessary for the computation of a single tile should fit into memory. From this figure, several conclusions can be drawn.

First, if an insufficient amount of concurrency is available, the OpenOOC runtime consistently results in increased runtime. For this benchmark problem, the OpenOOC system requires at least 64 fibers executing concurrently before the overhead introduced by the system is overcome by the performance benefits. To understand this result, take the bar in each page-cluster group for a single fiber. This bar quantifies the overhead associated with the OpenOOC system as a whole. For this benchmark, the effect of that overhead on runtime ranges from an 8–13%, with an average of 10%, increase when compared to the baseline (zero fibers). This is expected, since without sufficient

useful work to do, the system degrades to the baseline program, i.e., fibers waiting for swap operations to complete, with the additional overhead of context switching.

Another interesting observation from this figure is that increasing the page-cluster value only improves performance up to a certain point. This result is irrespective of the presence or absence of OpenOOC for small numbers of fibers, and has to do with the OS' swapping algorithms and the selection of tile size for the benchmark. In the Linux OS, the page-cluster value dictates how many consecutive pages are read from the swapfile or how many pages are written to consecutive locations in the swapfile. In the benchmark, the maximum length of a row within a tile is eight system pages, which is a consequence of the rest of the experimental design. In that case, if the page-cluster size is chosen to be greater than three, i.e., $2^3 = 8$ system pages, then irrelevant data that is consecutive in memory to the row being computed for, may be swapped in unnecessarily. Here by irrelevant data, we mean data that is part of the matrix, but not part of the tile. When the fiber count is small, OpenOOC is unable to overcome the cost of reading unnecessary data, just as with the baseline version. However, as the number of fibers is increased, the effect of the unnecessary data movement is lessened.

Finally, from Figure 7.1 we can analyze the absolute performance of the benchmark run with and without the OpenOOC system. For a fixed page-cluster size, the best configuration of the benchmark run with the OpenOOC system results in a 21% reduction in runtime on average when compared with the baseline application. This largest reduction is when page-cluster is five and results in a 29% reduction and the smallest is when page-cluster is three or four, which both result in a reduction of 15%. This makes sense since when page-cluster is three, the baseline application is reading all data for a partial dot-product in one swap and when it is equal to four it is likely reading the data for the next partial dot-product. Since the baseline application is not concurrently executing partial dot-products, the additional data being swapped, which will likely contain data for the next partial dot-product, is actually a benefit. It also makes sense that the largest improvement is when page-cluster equals five, since at this point, the baseline application will suffer from swapping unneeded data. When this happens, the OpenOOC system's ability to concurrently execute partial dot-products is especially important, since the largest page-cluster size means that the swap operation will take a much longer time. This is supported by the throughput results from Table 7.1. If we

look at the best performing configuration for the baseline application and the OpenOOC enabled application, we see a reduction in runtime with OpenOOC of 16%. This occurs for both applications when page-cluster is three. As discussed above, this is due to the experimental design and the choice of the tile size.

# Chapter 8

# Conclusion

## 8.1 Thesis Summary

### 8.1.1 In-Memory Compression-Based Methods

**Scientific Data Compression** In Chapter 3, we presented a paradigm for lossy compression of grid-based simulation data that achieves compression by modeling the grid data via a graph and identifying vertex-sets which can be approximated by a constant value within a user provided error constraint. Our comprehensive set of experiments showed that for structured and unstructured grids, these algorithms achieve compression which results in storage requirements that on average, are up to 75% lower than that other methods. Moreover, the near linear complexity of these algorithms makes them ideally suited for performing *in situ* compression in future exascale-class parallel systems.

**Dynamic Graph Compression** In Chapter 4, we presented five data structures for maintaining dynamically updated graphs. Three of the data structures are well known and treated as base line approaches. One of the data structures, DIAA, is optimized for memory usage and update and access time for classes of graphs which exhibit locality. The last data structure, DCAA, is an extension of DIAA, optimized entirely for reduced memory requirements. Each data structure was evaluated theoretically and experimentally using four real world datasets. The experimental analysis shows DIAA is superior

for web graphs and nearly as good as the others for social graphs. Further, our results show that, in all cases, DCAA is the most memory efficient data structure, at the cost of a modest increase in update and access time.

### 8.1.2 External Memory-Based Methods

**Distributed Memory Architecture** In Chapter 6, we presented three novel methods to automatically manage the virtual memory of parallel applications running on memory constrained systems, namely Aggressive Read / Aggressive Write, Aggressive Read / Lazy Write, and Lazy Read / Lazy Write. By leveraging the node-level cooperative multi-tasking constraints of BDMPI, we were able to optimize resident memory exchange to reduce data transfer to and from disk. For five different applications each exhibiting different memory access patterns, our results showed that SBMA offers performance gains between 2×–12× over applications executed using a BDMPI runtime that relies on the OS VMM to manage resident memory.

In a co-operative multi-tasking execution model, such as that used in BDMPI, once a process has been scheduled for execution, it runs, uninterrupted, until completion or it explicitly yields control of system resources. BDMPI exploits this characteristic to aggressively load / evict data to / from physical memory when processes transition from running to blocked. However, under certain circumstances, this policy of uninterruptable execution may result in idle processes due to pending inter-node communications. For example, suppose that some node, $i$, has a long running slave process, and another node, $j$, has slaves that are all waiting on communication from slaves on node $i$ other than the one running. In this case, all slaves on node $j$ will be blocked until the long running slave on node $i$ completes and some other slaves are allowed to execute.

In future work we will investigate improvements to the current BDMPI intra-node slave scheduler that address the issue of inter-node dependencies. One possibility is to maintain statistics related to the number of outstanding inter-node communications for each slave process and increase scheduling priority for those slaves with many outstanding inter-node communications. This is a variation of a coscheduling problem [92, 93] in concurrent systems. However, in our case, all of the slaves need not be scheduled together, but should be scheduled within a small window to reduce idle time. Still, we plan to leverage the work done in this area to guide any scheduling heuristics developed

for the BDMPI runtime. To judge the quality of our scheduling policies, we plan to characterize optimal or near-optimal orderings and compare them with the orderings imposed by our scheduling decisions in terms of process idle time.

**Shared Memory Architecture** In Chapter 7, we presented a novel runtime system to automatically switch user execution contexts in response to non-resident memory access. The new system relies on OpenMP to identify sections of code that express data-parallelism which is the ideal environment for using concurrency to hide long latency memory access. By leveraging the available concurrency, we were able to effectively reduce the amount of time that a program spends not doing useful work. A suite of micro-benchmarks was presented to demonstrate that modern OSs, Linux in particular, are capable of supporting fast enough context switches make a system like this feasible. For the computational problem of matrix-multiplication, which serves as a baseline because of its predictable memory access patterns, our results showed that without any effort on the part of the application developer, runtime can be decreased by 21% compared with an environment depending only on the latency hiding mechanisms common in modern OSs.

One challenge specific to automating out-of-core execution in a shared memory system is the sharing of data between multiple execution contexts in the runtime. In the distributed memory environment, i.e., BDMPI, execution contexts are OS processes. Thus, when a process enters the blocked state, since it does not share its address space with any other execution contexts, all of its data can be safely evicted from physical memory. In the shared memory runtime that we presented, all execution contexts will share a view of a single address space. Thus, the definition of what defines the data belonging to a particular execution context is not clear. In the future we plan to investigate policies for moving data between physical memory and secondary storage that exploit the constrained environment of our shared memory runtime for out-of-core computation and thus improve over the general purpose memory exchange policies implemented in modern OSs [94–99].

# Bibliography


[1] ARB OpenMP. Openmp 4.0 specification, june 2013, 2013.

[2] Tony Hey, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, October 2009.

[3] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

[4] Jeffrey Scott Vitter. algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2008.

[5] Angela Demke Brown. Explicit compiler-based memory management for out-of-core applications. Technical report, Stanford University, 2005.

[6] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Springer, 2009.

[7] D.R. Engler, S.K. Gupta, and M.F. Kaashoek. Avm: application-level virtual memory. In *Hot Topics in Operating Systems, 1995. (HotOS-V), Proceedings., Fifth Workshop on*, pages 72–77, May 1995.

[8] Brian Van Essen, Henry Hsieh, Sasha Ames, and Maya Gokhale. Di-mmap: A high performance memory-map runtime for data-intensive applications. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 731–735, Washington, DC, USA, 2012. IEEE Computer Society.

[9] Apache™Hadoop®. `http://hadoop.apache.org`.

[10] Chao-Hsien Lee, Meng Chang Chen, and Ruei-Chuan Chang. Hipec: High performance external virtual memory caching, 1994.

[11] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[12] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[13] Mitesh R Meswani, Gabriel H Loh, Sergey Blagodurov, David Roberts, John Slice, and Mike Ignatowski. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 9–16. IEEE, 2014.

[14] Y. Park, R. Scott, and S. Sechrest. Virtual memory versus file interface for large, memory-intensive scientific applications. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 53–53, 1996.

[15] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Jr. Joel Emer. Adaptive insertion policies for high performance caching. In *In Proceedings of the 35th International Symposium on Computer Architecture*, 2007.

[16] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 50:161–179, 1999.

[17] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Dimmap:a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, 2015.

[18] S.G. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[19] S. Muraki. Approximation and rendering of volume data using wavelet transforms. *Proceedings Visualization '92*, D:21–28, 1992.

[20] S. Muraki. Volume data and wavelet transforms. *IEEE Computer Graphics and Applications*, 13(4):50–56, July 1993.

[21] Amir Said and William A. Pearlman. A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Tress. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3):243–250, 1996.

[22] Jerome M. Shapiro. Embedded Image Coding Using Zerotrees of Wavelet Coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, 1993.

[23] Insung Ihm and Sanghun Park. Wavelet-Based 3D Compression Scheme for Interactive Visualization of Very Large Volume Data. *Computer Graphics Forum*, 18(1):3–15, March 1999.

[24] Ky Giang Nguyen and Dietmar Saupe. Rapid High Quality Compression of Volume Data for Visualization. *Computer Graphics Forum*, 20(3):49–57, September 2001.

[25] F.F. Rodler. Wavelet based 3D compression with fast random access for very large volume data. *Proceedings. Seventh Pacific Conference on Computer Graphics and Applications (Cat. No.PR00293)*, D:108–117.

[26] A. Skodras, C. Christopoulos, and T. Ebrahimi. The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine*, 18(5):36–58, 2001.

[27] Tallat M Shafaat and Scott B Baden. A Method of Adaptive Coarsening for Compressing Scientific Datasets. In *Applied parallel computing: State-of-the-Art in Scientific and Parallel Computing, 8th Intl. Workshop, Proc. PARA '06*, pages 1–7, 2006.

[28] Ricardo AF Belfor, Marc PA Hesp, Reginald L Lagendijk, and Jan Biemond. Spatially adaptive subsampling of image sequences. *IEEE Transactions on Image Processing*, 3(5):492–500, 1994.

[29] Didem Unat, Theodore Hromadka III, and Scott B. Baden. An Adaptive Subsampling Method for In-memory Compression of Scientific Data. *2009 Data Compression Conference*, pages 262–271, March 2009.

[30] Zachi Karni and Craig Gotsman. Spectral compression of mesh geometry. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*, pages 279–286, 2000.

[31] R Coifman and M Maggioni. Diffusion wavelets. *Applied and Computational Harmonic Analysis*, 21(1):53–94, July 2006.

[32] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The connectivity server: Fast access to linkage information on the web. *Computer networks and ISDN Systems*, 30(1-7):469–477, 1998.

[33] Keith H Randall, Raymie Stata, Rajiv G Wickremesinghe, and Janet L Wiener. The link database: Fast access to graphs of the web. In *Data Compression Conference, 2002. Proceedings. DCC 2002*, pages 122–131. IEEE, 2002.

[34] Micah Adler and Michael Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 203–212. IEEE, 2001.

[35] Paolo Boldi and Sebastiano Vigna. The webgraph framework ii: Codes for the world-wide web. Technical report, In DCC, 2003.

[36] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[37] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. Permuting web graphs. In *Algorithms and Models for the Web-Graph*, pages 116–126. Springer, 2009.

[38] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM, 2011.

[39] Torsten Suel and Jun Yuan. Compressing the graph structure of the web. In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 213–222. IEEE, 2001.

[40] Sriram Raghavan and Hector Garcia-Molina. Representing web graphs. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 405–416. IEEE, 2003.

[41] Yasuhito Asano, Yuya Miyawaki, and Takao Nishizeki. Efficient compression of web graphs. *Computing and Combinatorics*, pages 1–11, 2008.

[42] Francisco Claude and Gonzalo Navarro. A fast and compact web graph representation. In *International Symposium on String Processing and Information Retrieval*, pages 118–129. Springer, 2007.

[43] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[44] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 95–106. ACM, 2008.

[45] V. Engelson, D. Fritzson, and P. Fritzson. Lossless compression of high-volume numerical data from simulations. *Proceedings DCC 2000. Data Compression Conference*, page 574.

[46] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast Lossless Compression of Scientific Floating-Point Data. *Data Compression Conference (DCC'06)*, pages 133–142.

[47] Chuck Baldwin, Ghaleb Abdulla, and Terence Critchlow. Multi-resolution modeling of large scale scientific simulation data. *Proceedings of the twelfth international conference on Information and knowledge management - CIKM '03*, page 40, 2003.

[48] Free Software Foundation. Gzip implementation, 2017.

[49] Julian Seward. Bzip2 implementation, 2017.

[50] Igor Pavlov. Lzma implementation, 2017.

[51] G Karypis. METIS˜5.0: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 2011.

[52] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and Evolution of Online Social Networks. In *Link Mining: Models, Algorithms, and Applications*, pages 337–357. Springer, 2010.

[53] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, pages 343–350. acm, 2000.

[54] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

[55] Volker Stix. Finding all maximal cliques in dynamic graphs. *Comput. Optim. Appl.*, 27(2):173–186, February 2004.

[56] Michael A Bender and Haodong Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)*, 32(4):26, 2007.

[57] Georgia Mali, Panagiotis Michail, and Christos Zaroliagis. A new dynamic graph data structure for large-scale transportation networks. Technical report, eCompass, 2012.

[58] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[59] Alberto Apostolico and Guido Drovandi. Graph compression by bfs. *Algorithms*, 2(3):1031–1044, 2009.

[60] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In SuleymanCenk Sahinalp, S. Muthukrishnan, and Ugur Dogrusoz, editors, *Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 400–408. Springer Berlin Heidelberg, 2004.

[61] MPI: A Message-Passing Interface Standard Version 3.0. `www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012.

[62] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[63] The Open MPI Project. Open mpi, 2017.

[64] Argonne National Lab. Mpich, 2017.

[65] Cray. Cray message passing toolkit release overview, 2017.

[66] Microsoft. Microsoft mpi, 2017.

[67] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[68] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[69] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.

[70] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[71] Apache$^{TM}$Giraph. `http://giraph.apache.org`.

[72] Dominique LaSalle and George Karypis. Mpi for big data: New tricks for an old dog. *Parallel Computing*, 40(10):754–767, 2014.

[73] Yang suk Kee, Jin-Soo Kim, and Woo-Chul Jeun. Atomic page update methods for openmp-aware software dsm. In *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pages 144–151, Feb 2004.

[74] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[75] ParMetis. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[76] Shaden Smith and George Karypis. A medium-grained algorithm for distributed sparse tensor factorization. *30th IEEE International Parallel & Distributed Processing Symposium*, 2016, to appear.

[77] Stuart Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[78] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. SCI. COMPUT*, 14:654–669, 1992.

[79] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[80] Olaf Schenk, Andreas Wächter, and Martin Weiser. Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM Journal on Scientific Computing*, 31(2):939–960, 2008.

[81] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka, and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *In AAAI*, 2010.

[82] D. D. Lewis, Y. Yang, T. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *Machine Learning Research*, 5:361–397, 2004.

[83] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[84] CUDA Nvidia. Programming guide, 2008.

[85] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[86] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. *ACM SIGARCH Computer Architecture News*, 18(3b):1–6, 1990.

[87] Juan Jose Costa, Toni Cortes, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Running openmp applications efficiently on an everything-shared sdsm. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 35. IEEE, 2004.

[88] Todd C Mowry, Charles QC Chan, and Adley KW Lo. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 300–311. IEEE, 1998.

[89] Kritchalach Thitikamol and Pete Keleher. Multi-threading and remote latency in software dsms. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 296–304. IEEE, 1997.

[90] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[91] Greg Bronevetsky and Bronis R De Supinski. Complete formal specification of the openmp memory model. *International Journal of Parallel Programming*, 35(4):335–392, 2007.

[92] John K Ousterhout. Scheduling techniques for concurrebt systems. In *ICDCS*, volume 82, pages 22–30, 1982.

[93] Fabrizio Petrini and Wu-chun Feng. Improved resource utilization with buffered coscheduling. *PARALLEL ALGORITHMS AND APPLICATION*, 16(2):123–144, 2001.

[94] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[95] Richard W Carr and John L Hennessy. Wsclock — a simple and effective algorithm for virtual memory management. *ACM SIGOPS Operating Systems Review*, 15(5):87–95, 1981.

[96] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[97] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[98] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[99] Andrew Tanenbaum. Modern operating systems. 2009.

[100] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228. ACM, 2009.

[101] U. Kang and Christos Faloutsos. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, ICDM '11, pages 300–309, Washington, DC, USA, 2011. IEEE Computer Society.

# Appendix A

# WebGraph Compression

The WebGraph format [36] employs the following three techniques. First, reference compression is used to leverage the similarity of adjacency information. This means that instead of representing the adjacency list of a particular vertex $u$ directly, they represent it as a reference to another similar vertex $v$. This is done using a bit mask of length $|\delta(v)|$. A 1-bit at index $i$ of the bit mask signifies that vertex $u$ has a link to the $i^{th}$ link in vertex $v$'s adjacency list, while a 0-bit signifies that vertex $u$ has no such link. This bit mask can then be run-length encoded as 1-blocks and 0-blocks, the details of which can be found in [36].

After reference compression of $\Gamma(u)$, an intervalization technique is applied. Any vertex ids in $\Gamma(u)$ which did not appear in $\Gamma(v)$ are represented as a set of intervals and a list of residual ids. An *interval* is a consecutive sequence of vertex ids, which are common assuming locality of adjacency lists. Each interval is encoded as a start id and a length, so the sequence of vertex ids $3, 4, 5, 6, 7, 8$, would be encoded as $(3, 6)$. Any vertex ids of $\Gamma(u)$ which did not appear in $\Gamma(v)$ or any interval is encoded in the residual list. The start ids of each interval is differentially encoded, i.e., stored as the difference from the previous start id. Differential encoding is performed to decrease the range of integer values, which is beneficial to the integer coding scheme described below.

Any vertex ids of $\Gamma(u)$ which did not appear in $\Gamma(v)$ or any interval is encoded in the residual list. This is simply a list of vertex ids left over from the previous two steps and is differentially encoded. Since the difference between vertex id and the first interval start id or first residual may be negative, and the integer coding described below

operates on non-negative integers only, a mapping $v : \mathbb{Z} \to \mathbb{N}$ is established according to the following rule:

$$v(x) = \begin{cases} 2x & \text{if } x >= 0 \\ 2|x| - 1 & \text{if } x < 0 \end{cases}$$

To code the integers in the above scheme, the authors of [36] developed an instantaneous integer coding algorithm, suitable for storage of integers which exhibit power-law distribution, which the authors showed was the case for encodings of this type. Their integer code is called $\zeta$-codes and is formally described in [35].

The above discussion leverages the ability to lexicographically sort the vertices of web graphs based on their URLs to expose locality and similarity. However, in the case of online social graphs, lexicographically sorting the vertices does not guarantee the ability to expose locality and similarity. Thus, it is necessary to develop ordering schemes for online social networks which are conducive to this type of compression. Many orderings have been developed along these lines including BFS [59], Grey [37], Shingle [100], SlashBurn [101], however it was shown in [38] that all are inferior to the authors own algorithm called Layered Label Propagation (LLP). In their algorithm, the graph is repeatedly clustered and each iteration, the vertices are reordered so that vertices within a cluster are given consecutive ids. The clustering algorithm that is employed in LLP allows for clustering to be performed at different 'resolutions', such that each iteration of clustering will yield clusters which capture more or less specific characteristics of the graph under consideration. This allows the LLP to increase resolution each iteration and in this way, iteratively 'refine' the ordering of vertices in the graph.

# Appendix B

# Subset of MPI API

Table B.1: The subset of the MPI API implemented by BDMPI. [72]

| |
|---|
| MPI_Init, MPI_Finalize |
| MPI_Comm_size, MPI_Comm_rank, MPI_Comm_dup, MPI_Comm_free, MPI_Comm_split |
| MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv, MPI_Sendrecv |
| MPI_Probe, MPI_Iprobe, MPI_Test, MPI_Wait, MPI_Get_count |
| MPI_Barrier |
| MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Scan, MPI_Gather[v], MPI_Scatter[v], MPI_Allgather[v], MPI_Alltoall[v] |

# Appendix C

# Subset of OpenMP API

Table C.1: A subset of the directives defined by OpenMP, along with a brief description of their purpose. [1]

| Directive | Description |
|---|---|
| PARALLEL | Defines a parallel region. |
| FOR | Identifies an iterative worksharing construct in which the iterations of the associated loop should be divided among threads in a team. |
| SECTIONS | Identifies a non-iterative worksharing construct that specifies a set of structured blocks that are to be divided among threads in a team. |
| SECTION | Indicates that the associated structured block should be executed in parallel as part of the enclosing sections construct. |
| SINGLE | Identifies a construct that specifies that the associated structured block is executed by only one thread in the team. |
| MASTER | Identifies a construct that specifies a structured block that is executed by only the master thread of the team. |
| CRITICAL | Identifies a construct that restricts execution of the associated structured block to a single thread at a time. Each thread waits at the beginning of the critical construct until no other thread is executing a critical construct with the same name argument. |

continued on the next page

Table C.1: A subset of the directives defined by OpenMP, along with a brief description of their purpose. [1] (cont.)

| Directive | Description |
|---|---|
| BARRIER | Synchronizes all the threads in a team. Each thread waits until all of the other threads in that team have reached this point. |
| ATOMIC | Ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneously writing threads. |

Table C.2: A subset of the clauses defined by OpenMP, along with a brief description of their purpose. [1]

| Clause | Description |
|---|---|
| PRIVATE (list) | Declares variables in list to be PRIVATE to each thread in a team. |
| NOWAIT | Specifies that threads need not wait at the end of worksharing constructs until they have completed execution. The threads may proceed past the end of the worksharing constructs as soon as there is no more work available for them to execute. |
| SHARED (list) | Shares variables in list among all the threads in a team. |
| IF (expr) | The enclosed parallel region is executed in parallel only if expr evaluates to TRUE, otherwise the parallel region is serialized. The expression must be scalar logical. |
| NUM_THREADS (expr) | Requests the number of threads specified by expr for the parallel region. The expressions must be scalar integers. |
| SCHEDULE (type) | Specifies how iterations of the FOR construct are divided among the threads of the team. Possible values for the type argument are STATIC, DYNAMIC, GUIDED, and RUNTIME. |
| REDUCTION (reduction-id:list) | Specifies a reduction-id and one or more list items. The reduction-id must match a previously declared reduction-identifier of the same name and type for each of the list items. |

# Appendix D

# Example OpenMP function

Listing D.1: An OpenMP implementation of a vector dot-product.

```
1  double dot_product(int n, double * u, double * v)
2  {
3    double sum = 0.0;
4    #pragma omp parallel reduction(+:sum)
5    {
6      #pragma omp for
7      for (int i=0; i<n; ++i)
8        sum += u[i] * v[i];
9    }
10   return sum;
11 }
```